

Contributions to the Decoding and Structure of Convolutional Codes

Mats Cedervall

Lund 1983



Contributions to the Decoding and Structure of Convolutional Codes

AKADEMISK AVHANDLING

som för avläggande av teknisk doktorsexamen vid tekniska fakulteten
vid Universitetet i Lund kommer att offentligen försvaras i hörsal E:B,
Lunds Tekniska Högskola, måndagen den 7 november 1983 kl 10.15

av

Mats Cedervall

civ.ing.

Organization LUND UNIVERSITY Avd. för Digital- och Datorteknik, LTH Box 725, 220 07 LUND, Tel 046-107515 Dept. of Computer Engineering, Univ. of Lund, PO Box 725, S-220 07 LUND, Sweden		Document name DOCTORAL DISSERTATION	
		Date of issue Nov 1983	
		CODEN: LUTEDX/(TEDD-1004)/1-110/(1983)	
Author(s) Mats Cedervall		Sponsoring organization	
Title and subtitle CONTRIBUTIONS TO THE DECODING AND STRUCTURE OF CONVOLUTIONAL CODES			
Abstract Error control coding is often used to increase the efficiency of data communication systems. This is due to the dramatic improvement in error control techniques and the increased use of satellite links. There are basically two different approaches to the problems of constructing good codes. The algebraists choose block codes with suitable mathematical structure to simplify the development of decoding algorithms. Rather than dealing with specific codes the probabilists specify decoding algorithms which are efficient procedures for certain ensembles of randomly selected tree codes. Convolutional codes constitute the class of linear, time-invariant tree codes. In almost every error correcting application convolutional codes are superior to block codes. This thesis presents methods for determining quality parameters for convolutional codes, viz. the decoding error probability and different distances. We give bounds on decoding error probability which are tighter than previously known bounds. These bounds are useful when we compare different codes with each other. FAST algorithms for computing distance spectrum parameters are presented. They are easily realized as computer programs. Finally, a structured way of specifying a sequential machine is described.			
Key words Convolutional codes, fast tree algorithms, free distance, ODP, minimum distance			
Classification system and/or index terms (if any) Electrical Engineering, Information theory			
Supplementary bibliographical information			Language English
ISSN and key title			ISBN
Recipient's notes	Number of pages 110	Price	
	Security classification		

Distribution by (name and address)

I, the undersigned, being the copyright owner of the abstract of the above-mentioned dissertation, hereby grant to all reference sources permission to publish and disseminate the abstract of the above-mentioned dissertation.

Signature _____

Date 23 sep 1983 _____

Contributions to the Decoding and Structure of Convolutional Codes

Mats Cedervall



Lund 1983

Contributions to the Decoding and Structure of Convolutional Codes

Mats Cedervall



© Mats Cedervall 1983

Printed in Sweden

Studentlitteratur

Lund 1983

Lund 1983

INNEHÅLL

TILL PIA OCH JOHAN

INNEHÅLL

1. BAKGRUNDSUPPLÄGGNING

1.1. Bakgrund	1
1.2. Syfte och mål	10
1.3. Begränsningar	10
1.4. Översikt över rapportens innehåll	10

2. A BAKGRUND I BEHÅLLNINGEN AV FÖRSTÄLLNINGAR OM HÅLLBARHET FÖR HÅLLBARHET OCH HÅLLBARHET OM FÖRSTÄLLNINGAR OM HÅLLBARHET

2.1. Inledning	11
2.2. Förstärkt utvärdering - Hållbarhet	11
2.3. Förstärkt utvärdering - Hållbarhet	11
2.4. A BAKGRUND I BEHÅLLNINGEN	11
2.5. A BAKGRUND I BEHÅLLNINGEN	11
2.6. A BAKGRUND I BEHÅLLNINGEN	11
2.7. A BAKGRUND I BEHÅLLNINGEN	11

3. ÖVERSIKT ÖVER HÅLLBARHET

CONTENTS

PREFACE	7
1 INTRODUCTORY CONCEPTS	9
1.1 Introduction	9
1.2 Convolutional encoding	11
1.3 Group property of zero-delay polynomials	16
1.4 How to generate systematic ODP encoders	18
2 A NEW UPPER BOUND ON THE FIRST-EVENT ERROR PROBABILITY FOR MAXIMUM-LIKELIHOOD DECODING OF FIXED BINARY CONVOLUTIONAL CODES	21
2.1 Introduction	21
2.2 Many channel errors - RANDOM WALK	22
2.3 Few channel errors - UNION BOUND	26
2.4 A tighter Viterbi bound	28
2.5 A tighter Van de Meeberg type bound	30
2.6 Generating function	32
2.7 Numerical results	33
3 COMPUTING DISTANCE SPECTRUM	41

3.1	Problem	41
3.2	A BASIC algorithm	42
3.3	An IMPROVED basic algorithm	47
3.4	Applying a stack	50
3.5	A FAST algorithm	52
3.6	Modifications of the FAST algorithm	58
3.7	Results	62
3.8	Conclusions	63
4	CDC, A REALIZATION OF FAST	65
4.1	General problem	65
4.2	Partitioning CDC	66
4.3	The path counter	68
4.4	The branch length counter	70
4.5	The distance profile	71
4.6	The actual node	72
4.6.1	The actual state	73
4.6.2	The actual weight	75
4.7	The stack	82
4.8	The Central Scrutinizer	87
4.9	Conclusions	100
	APPENDIX: RESULTS, Convolutional codes	101
	REFERENCES	107

PREFACE

Error control coding is more often used to increase the efficiency of data communication systems. This is due to the dramatic improvement in error control techniques and the increased use of satellite links.

There are basically two different approaches to the problems of constructing good codes: block codes and convolutional codes. In almost every error correcting application convolutional codes are superior to block codes.

This thesis presents methods for determining quality parameters for convolutional codes, viz. the decoding error probability and different distances.

In Chapter 1 we give an introduction to the subject and present an efficient algorithm for extending systematic optimum distance profile (ODP) codes. In an appendix we list the best systematic ODP codes up to memory 96.

New tighter upper bounds on the error probability are reported in Chapter 2. This part is a joint work together with Rolf Johannesson and Kamil Sh. Zigangirov. This research was presented at the IEEE International Symposium on Information Theory in Les Arcs, France, 1982 and a slightly revised version of this chapter will appear in IEEE Transactions on Information Theory.

Distance parameters are the principal determiner of decoding error probability. Hence, in Chapter 3 we address the problem of computing distance parameters. A FAST tree algorithm is developed from a basic algorithm. These results were partly obtained together with Rolf Johannesson and presented both at the first Swedish-Soviet Workshop on Convolutional Codes and Multi-user communication at Sochi, USSR, 1983 and at the IEEE International Symposium on Information Theory in St. Jovite, Canada, 1983. New results on the best known nonsystematic ODP codes are given in an appendix.

Finally, Chapter 4 describes a structured method to specify a sequential machine. The method is exemplified by a realization of FAST in hardware - the birth of the Column Distance Checker, CDC. This method is a graph approach to a solution of the state machine problem.

Now it is time for the superlatives. I would like to thank **Christer Fernström** who gave me a special version of the famous ASMEDIT, **Anders Ardö** and his SubScribe allowing me to publish this material in my way, **Kerstin Berglund** and **Don Speck** for reading parts of my thesis. A picture may not be replaced by thousands of words. Due to this I would specially thank **Kerstin Johnson** and **Victor A. Xenakis** for their drawings. For constant motivation and suggestions when we prepared the outlines of Chapter 2, I would like to thank **Kamil**. Encouragement and support excludes depressions when working for a doctor's degree. The person who brought me into my work has extraordinarily fulfilled this. A flow of superlatives would not give justice to him. **Thank you Rolf!** Since this thesis was mostly prepared out-of-hours, i.e. in the spare time which should have been allocated to my family, my wife **Pia's** contribution is twofold: She has been taking care of me and our son **Johan** during this period and furthermore reading and correcting errors in my material. Therefore I am especially grateful to her. Finally, I hope that this thesis will contribute to the correcting of the error: Johan is calling me "mamma".

INTRODUCTORY CONCEPTS

1.1 INTRODUCTION

The ability and desire to communicate are two of the most considerable features in our nature. Consistently we must guarantee this flow of information so that we can rely upon it. Reliability is desirable. Or, as stated by the father of **information theory**, Claude E. Shannon (Shannon, 1948):

"The fundamental problem of communication is that of reproducing at one point either exactly or approximately a message selected at another point."

Shannon's work started a new movement in the field of communication and inspired many engineers to contribute with research in information theory.

Let us assume that we can form a message as one of two symbols, 0 and 1, having the same probability. To deliver a message, say 0, to the receiver, we must pass it through some kind of channel. During this transportation the message has a tendency to change (that's nature), 0 sometimes becomes 1 and vice versa. An **error** occurs.

The receiver will also accept messages in error because they are valid messages.

A straight forward method to decrease the error probability would be to improve the channel. But, as said by Richard E. Blahut (Blahut, 1983):

"To build a communication channel as good as we can is a waste of money."

Instead we will apply a scheme of rules, **error control** (=error detection and/or correction), between the transmitter and the receiver.

To improve the receiver's certainty, we can simply transmit the symbol twice. In this case the receiver is able to **detect** when one error has happened during the transportation, because the two messages 01 and 10, each containing one error, are not in the set of permitted transmittable messages, 00 and 11. Once an error has been detected, the receiver can tell the transmitter that something is wrong, and that the message should be retransmitted. This feedback from the receiver to the transmitter is not always possible, because there may be no channel in this direction. We have to operate better.

Continuing, we could repeat the symbol three times. Now the receiver is able to **correct** a received message containing one error. If it receives 100, 010 or 001, it just makes a majority decision and decides 0.

If we expand this method we see that a message made of d , where $d = 2t+1$, repeated symbols implies the ability to correct t errors or to detect $2t$ errors at the receiver. The disadvantage of this method is that for each information symbol (0 or 1) we have to send d symbols over the channel. Consequently it takes d times as long to send a message. In other words, if the channel is capable of transmitting one symbol each time unit, then the source may only generate messages with the **rate** $R = 1/d$. This is against our wishes.

Will safety be contradictory to rate? Fortunately it need not to be. If the selected rate is below a certain rate C called the **channel capacity**, the probability that the receiver will make an error can be made arbitrary small, by increasing the information length on which we operate. We trade complexity for reliability. This result is due to Shannon and is called the **channel coding theorem**.

The coding can be done in two different ways, **block-** and **tree-coding**. Block coding is done by taking K information symbols and map those to N channel symbols ($K < N$) in some special way. Send these new N channel symbols, take K new information symbols, generate N new channel symbols and so on. In this case the rate is $R = K/N$. As an example we have the Hamming codes (Hamming, 1950) which correct all single errors within a block. If $K = 4$ then for the Hamming codes we have $N = 7$ and the rate will be $R = 4/7 > 1/3$ as for the repetitive case. More powerful codes than the Hamming codes were found by Reed and Solomon (Reed et al, 1960). These RS codes still remain among the most important classes of block codes and they are e.g. used for correcting errors on compact discs in digital record players.

While block codes have a strong algebraic flavor, tree codes are often used together with probabilistic decoding algorithms. The most useful tree codes are highly structured codes called **convolutional codes**. Convolutional coding means that the channel symbols are generated as "sliding" sums of the information symbols, convolving the information sequence with a polynomial, the code generator. The current channel symbol is generated from the M latest information symbols and the current one. M is called the **memory** of the encoder.

1.2 CONVOLUTIONAL ENCODING

Definition: A general (N,K) convolutional encoder G is an invertible K -input, N -output realizable linear finite state machine (FSM) which is in its zero state in the infinite past.

Realizable means that the encoder output cannot depend upon future inputs. **Invertible** (Massey et al, 1967) means that the input can be recovered, perhaps with delay, from the encoder output. If G is **time-invariant**, it is called a **fixed convolutional encoder**. In this thesis we will only deal with rate one-half binary fixed convolutional encoders. See Figure 1.2.1.

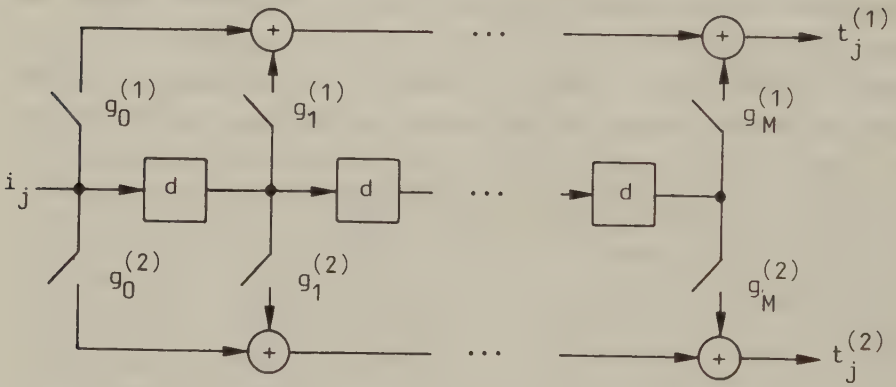


Figure 1.2.1 A model of an $R = 1/2$ binary encoder.

The information sequence $I = i_0, i_1, i_2, \dots$ is encoded as the sequence

$$T = t_0^{(1)}, t_0^{(2)}, t_1^{(1)}, t_1^{(2)}, t_2^{(1)}, t_2^{(2)}, \dots \quad (1.2.1)$$

where

$$t_j^{(k)} = \sum_{n=0}^M i_{j-n} g_n^{(k)} \quad (1.2.2)$$

and

$$G^{(k)} = [g_0^{(k)}, g_1^{(k)}, \dots, g_M^{(k)}] \quad (1.2.3)$$

$k = 1, 2$ are the code **generators**. When $G^{(1)} = [1, 0, \dots, 0]$ the encoder is called **systematic**.

It is also possible to specify the generators as polynomials using a "dummy" variable D whose task is to indicate the actual place of a connection:

$$G^{(k)}(D) = g_0^{(k)} + g_1^{(k)}D + \dots + g_M^{(k)}D^M \quad (1.2.4)$$

Using this notation the generators of the encoder in Figure 1.2.2 are $G^{(1)}(D) = 1+D+D^2+D^3$ and $G^{(2)}(D) = 1+D^2+D^3$.

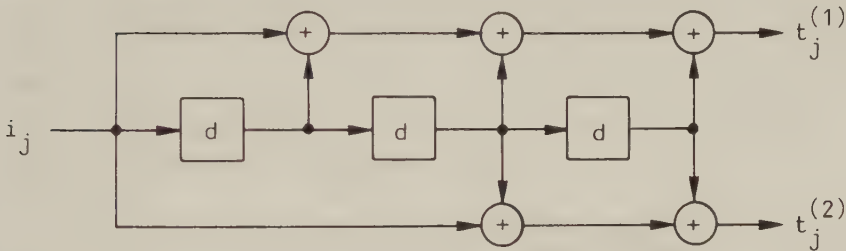


Figure 1.2.2 An example of an $R = 1/2$ $M = 3$ encoder.

Following (Johannesson, 1975) we use an octal notation for generators where the first octal digits denotes $[g_0^{(k)}, g_1^{(k)}, g_2^{(k)}]$, the second denotes $[g_3^{(k)}, g_4^{(k)}, g_5^{(k)}]$, etc. Thus, we write the generators in Figure 1.2.2 as 74 and 54. When we extend generators it is obvious from this notation that the original generator is a truncation of the extended one.

Let us define the **state** S_j of an encoder as an M -tuple consisting of the M latest information symbols.

$$S_j = [i_{j-1}, i_{j-2}, \dots, i_{j-M}] \quad (1.2.5)$$

Knowing the state S_j and the present information symbol i_j , the **next state** S_{j+1} is given by

$$S_{j+1} = \delta(S_j, i_j, M) = [i_j, i_{j-1}, \dots, i_{j-M+1}], \quad (1.2.6)$$

where δ is called the next state function and the third parameter specifies the number of state variables in S_{j+1} .

If there exists a state S and an information sequence I , not equal to the all zero sequence, such that when applying I to the encoder in state S , we end up in the same state S , and the encoded sequence is the all zero sequence, we have a **catastrophic** encoder. A theorem of (Massey et al, 1968) states that a fixed convolutional code is catastrophic if and only if the generators have a common factor not equal to a delay. As an example, assume we have the code $G(D) = [1+D^3, 1+D+D^2+D^3]$. The generators have the common factor $P(D) = 1+D$. Then we have a new realization according to Figure 1.2.3.

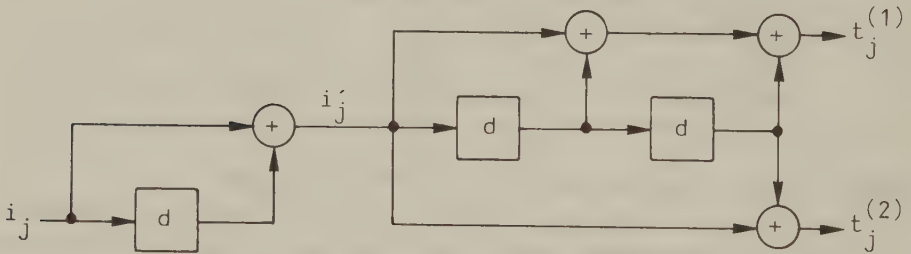


Figure 1.2.3 An example of a catastrophic encoder.

It is obvious that there exists such an information sequence I , that produces a new sequence I' equal to the all zero sequence, after some time. In this case it will be $I = 1,1,1,\dots$ and hence $I' = 1,0,0,\dots$

It is convenient to write

$$T_n = [t_0^{(1)}, t_0^{(2)}, t_1^{(1)}, t_1^{(2)}, \dots, t_n^{(1)}, t_n^{(2)}] \quad (1.2.7)$$

for the encoded path containing the first $n + 1$ "branches" of the encoded sequence. The truncated channel sequence T_M is called the **first constraint length** of the code.

Definition: The **Hamming distance**, $d_H[T^{(1)}, T^{(2)}]$, between two sequences, $T^{(1)}$ and $T^{(2)}$, is the number of positions in which they differ.

Example $T^{(1)} = 1011010$, $T^{(2)} = 0111101$
 $d_H[1011010, 0111101] = 5$

The j -th order **column distance** d_j (Costello, 1969) is the smallest Hamming distance between some T_j resulting from an information sequence with $i_0 = 0$ and some T_j with $i_0 = 1$. By linearity, d_j is also the minimum Hamming weight of the paths T_j resulting from information sequences with $i_0 = 1$.

The quantity d_M is called the **minimum distance** of the convolutional code and determines the guaranteed error-correcting capability when the code is decoded by a "feedback decoder" (Massey, 1963). The quantity d_∞ is called the **free distance** of the code and has been found to be the principal determiner of decoding error probability when maximum-likelihood decoding is used.

The **distance profile** (Johannesson, 1975) of a code is defined as the $(M+1)$ -tuple

$$\mathbf{d} = [d_0, d_1, \dots, d_M]. \quad (1.2.8)$$

Comparing two distance profiles, $\mathbf{d}$ and $\mathbf{d}'$, $\mathbf{d}$ is said to be superior to $\mathbf{d}'$ if there exists some j such that,

$$d_k \left\{ \begin{array}{l} = d'_k, \quad k = 0, 1, \dots, j-1 \\ > d'_k, \quad k = j. \end{array} \right. \quad (1.2.9)$$

A code with memory M belongs to the set of codes with an **optimum distance profile**, denoted $ODP(M)$, if and only if its distance profile is equal to or superior to that of any code with the same memory. The encoder in Figure 1.2.2 belongs to $ODP(3)$.

1.3 GROUP PROPERTY OF ZERO-DELAY POLYNOMIALS.

Code generators $G^{(k)}$ with $g_0^{(k)} = 1$ (zero-delay) are of particular interest because they have the property that the two possible sets of branch digits which can be generated (the new information digit entering the encoder is either a 0 or a 1) must be binary complements.

Let $I^{(n)}$, $n = 1, 2, \dots, 2^M$, and $G^{(k)}$, $k = 1, 2$, be two binary zero-delay $(M+1)$ -tuples, whose j -th digits are $i_j^{(n)}$ and $g_j^{(k)}$, respectively. The convolution of $I^{(n)}$ and $G^{(k)}$ modulo D^{M+1} , denoted $I^{(n)} * G^{(k)}$, is defined as the binary $(M+1)$ -tuple $T_M^{(n,k)}$, whose j -th digit, $t_j^{(n,k)}$, is given by

$$t_j^{(n,k)} = i_0^{(n)} g_j^{(k)} + i_2^{(n)} g_{j-1}^{(k)} + \dots + i_j^{(n)} g_0^{(k)} \quad (1.3.1)$$

Since both $I^{(n)}$ and $G^{(k)}$ are zero-delay, $T_M^{(n,k)}$ is also zero-delay. The set of all the possible zero-delay polynomials forms an Abelian group under the binary convolution (Busgang, 1965). Hence, the same set of truncated sequences, $[T_M^{(1,1)}, T_M^{(1,2)}, T_M^{(2,1)}, \dots, T_M^{(m,2)}]$, where $m = 2^M$, generated by the 2^M different information sequences with $i_0^{(n)} = 1$ and $G = [G^{(1)}, G^{(2)}]$ could also be generated with $G^{(1)} = 1$, except for a rearrangement of the information sequences. In other words, the code $G = (G^{(1)}, G^{(2)})$ and the systematic code $G = (1, G_{\text{sys}})$, where $G^{(2)} = G^{(1)} * G_{\text{sys}}$, generate the same set of truncated sequences. If G is given by Figure 1.2.2, then one of the corresponding systematic codes is shown in Figure 1.3.1.

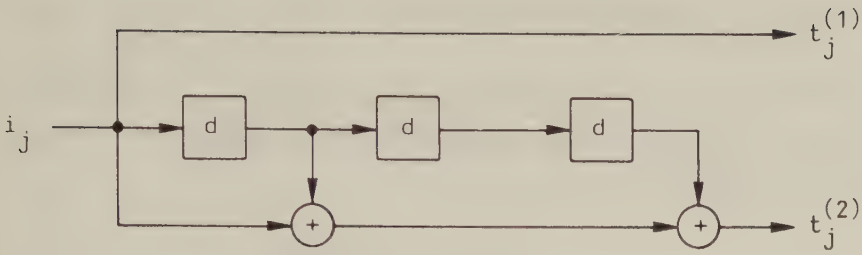


Figure 1.3.1 An example of a systematic ODP encoder.

Furthermore, the **adjoint code**, i.e. the code $G = (GA, 1)$, where $GA * G_{\text{sys}} = 1$, will generate the same set of truncated sequences.

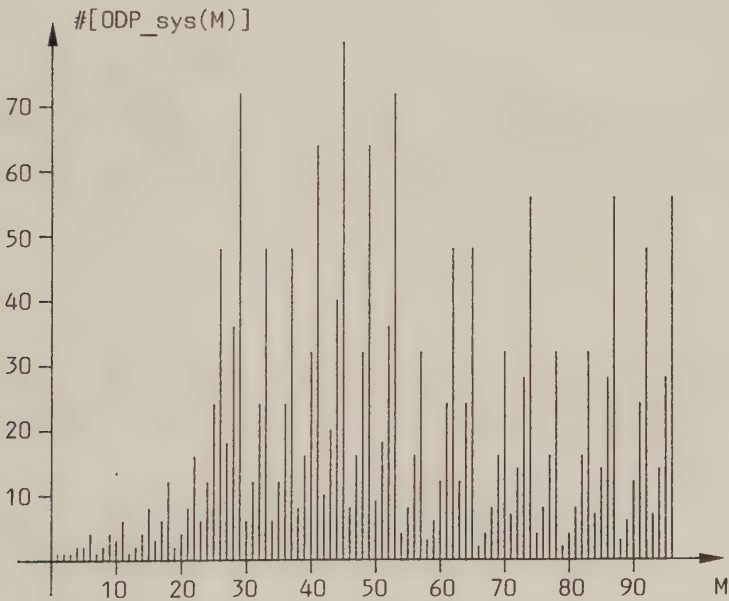


Figure 1.3.2 The number of systematic ODP convolutional codes.

The number of ODP-codes of a given memory M will be given by (1.3.2).

$$\#[\text{ODP}(M)] = \#[\text{ODP_sys}(M)] 2^M \quad (1.3.2)$$

where $\text{ODP_sys}(M)$ is the set of systematic ODP-codes and $\#[.]$ denotes the cardinality of a set. The number $\#[\text{ODP_sys}(M)]$ is shown in Figure (1.3.2) as a function from M .

1.4 HOW TO GENERATE SYSTEMATIC ODP ENCODERS.

Let us define the generator $G_n^{(2)}$ as the **n-extended** $G^{(2)}$ generator, where $n = 0, 1$:

$$G_n^{(2)} = [g_0^{(2)}, g_1^{(2)}, \dots, g_M^{(2)}, n]. \quad (1.4.1)$$

Furthermore, we need the set of states $S(G^{(2)}, M)$ consisting of the states that were produced by those information sequences that start with $i_0 = 1$ and $G^{(1)} = 1$, and correspond to encoded sequences with a minimum distance d_M . Such a state has the following form

$$S = [i_M, i_{M-1}, \dots, i_1, 1]. \quad (1.4.2)$$

Note that the number of state variables of those states are $M+1$ and $\#[S(G^{(2)}, M)]$ gives the number of paths having this minimum distance d_M .

Finally, we are ready to formulate an algorithm that will generate the set of $\text{ODP_sys}(M+1)$ codes from the set of $\text{ODP_sys}(M)$ codes:


```

for every generator  $G^{(2)}$  in  $ODP\_sys(M)$  do
  begin
    for each state  $S$  in  $S(G^{(2)}, M)$  do
      begin
        with  $i=0$  do
          if  $t^{(2)}=0$  then
             $S(G_0^{(2)}, M+1) \ni \delta(S, 0, M+2)$ 
          else
             $S(G_1^{(2)}, M+1) \ni \delta(S, 0, M+2)$ 
          endif
        end
      end
      if any  $\# [S(G_n^{(2)}, M+1)] = 0$  then
         $d_{M+1} = d_M + 1$ 
        for those  $G_n^{(2)}$  do
          begin
            calculate  $S(G_n^{(2)}, M+1)$  using a tree search
          end
        end
      else
         $d_{M+1} = d_M$ 
      endif
    end
  end

```

Algorithm 1.4.1 Extending $ODP_sys(M)$ to $ODP_sys(M+1)$.

Since the minimum distance d_M rarely changes, this algorithm will significantly reduce the computation time, for extending ODP-codes. Figure 1.4.1 shows d_M as a function of M .

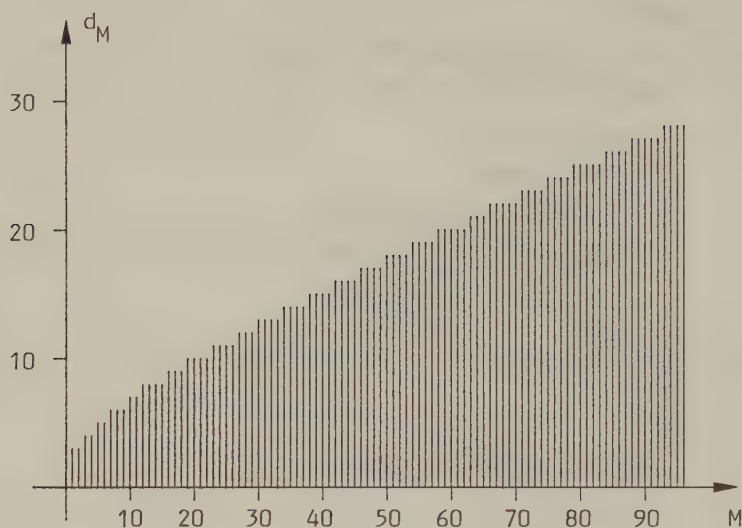


Figure 1.4.1 Minimum distance for ODP-codes.

An extensive table of systematic ODP codes is given in Appendix. Using the results in Section 1.3 we generated the complete set of nonsystematic ODP codes from the complete set of systematic ODP codes. In Appendix we also give a table of some good nonsystematic ODP codes together with the first parameters of their distance spectra.

A NEW UPPER BOUND ON THE FIRST-EVENT ERROR PROBABILITY FOR MAXIMUM-LIKELIHOOD DECODING OF FIXED BINARY CONVOLUTIONAL CODES

2.1 INTRODUCTION

Several authors have addressed the problem of analyzing the performance of a maximum likelihood (ML) decoder for convolutional codes. Viterbi (Viterbi, 1971) made clever use of signal flowchart techniques to derive his famous upper bound on the first-event error probability of ML decoding of fixed convolutional codes. Van de Meeberg (Meeberg, 1974) obtained a significant improvement of Viterbi's bound for large signal to noise ratios, while Post (Post, 1980) using quite a different technique derived a bound which is slightly better than these two bounds for low signal to noise ratios. Furthermore, Post (Post, 1977) showed how to calculate the union bound exactly. Finally, Schalkwijk et al (Schalkwijk et al, 1979) presented a method for exact calculation of the event error probability. However, this method is feasible only for rather short codes.

In this paper we derive tighter upper bounds of both Viterbi and Van de Meeberg type for binary convolutional codes on the binary symmetric channel (BSC). Our bound is significantly better than Van de Meeberg's bound for rates above the computational cut-off rate, R_{comp} .

Viterbi used the union bound in his derivation. While the union bound is quite tight when there are few channel symbols in error it is rather loose when we have many errors among the channel symbols. Thus, let us separate the error event into two disjoint events corresponding to few (F) and many (M) errors, respectively. Then, if we let E denote the event that the first information symbol is erroneously decoded by maximum-likelihood decoding on the binary symmetric channel (BSC), we have:

$$\begin{aligned}
 P[E] &= P[E|F]P[F] + P[E|M]P[M] \\
 &\leq P[E|F]P[F] + P[M] \\
 &= P[E,F] + P[M].
 \end{aligned}
 \tag{2.1.1}$$

In Section 2.2 we will use a random walk argument to upper bound the probability that we have many channel errors, $P[M]$. The union bound is used in Section 2.3 to obtain a good upper bound on the probability that the first information symbol is erroneously decoded and that we have few channel errors, $P[E,F]$. These two bounds are combined in Section 2.4 to a tighter Viterbi type bound, and in Section 2.5 we give an improved Van de Meeberg type bound. In Section 2.6 we show how to calculate the generating function of the output sequence lengths and weights. Finally, in Section 2.7 we discuss some numerical results.

2.2 MANY CHANNEL ERRORS - RANDOM WALK

To obtain an upper bound on the probability that we have many channel errors, $P[M]$, we use a random walk argument which is based on a little lemma given in (Gallager, 1968, p. 312):

Let $z_1, z_2, \dots$ be a sequence of statistically independent identically distributed discrete random variables. Then for any $\lambda \leq 0$ such that

$$E[2^{\lambda z_i}] \leq 1, \quad (2.2.1)$$

and for any f ,

$$P[\min_n \sum_{i=1}^n z_i \leq -f] \leq 2^{\lambda f}. \quad (2.2.2)$$

Let us introduce a metric s_e when we have a channel error and a metric s_c when the channel symbol is correctly received. Then, the metrics along the correct path is a random walk with $P[z_i = s_e] = p$ and $P[z_i = s_c] = q = 1-p$, where p is the cross-over probability of the BSC.

If we choose

$$s_e = \log \frac{p}{a} \quad (2.2.3)$$

and

$$s_c = \log \frac{q}{1-a}, \quad (2.2.4)$$

where $p < a < 1$ is a parameter to be chosen later, then inequality (2.2.1) gives

$$p\left(\frac{p}{a}\right)^\lambda + q\left(\frac{q}{1-a}\right)^\lambda \leq 1. \quad (2.2.5)$$

We notice that $\lambda = -1$ satisfies the inequality (2.2.5) and, hence, we have

$$P[\min_n \sum_{i=1}^n z_i \leq -f] \leq 2^{-f}. \quad (2.2.6)$$

Furthermore, it is interesting to notice that our metrics are closely

related to the Fano metric (Fano, 1963). Let φ_0 be the solution of

$$\varphi R = E_0(\varphi), \quad (2.2.7)$$

where $E_0(\varphi)$ is the Gallager function. If we choose a particular value of a , viz.

$$a = p^{1/(1+\varphi_0)} / (p^{1/(1+\varphi_0)} + q^{1/(1+\varphi_0)}) \quad (2.2.8)$$

(c.f. (Gallager, 1968, p. 146)), we obtain

$$s_e = \varphi_0 / (1+\varphi_0) (\log 2p - R) \quad (2.2.9)$$

and

$$s_c = \varphi_0 / (1+\varphi_0) (\log 2q - R), \quad (2.2.10)$$

which are off only by a factor $\varphi_0/(1+\varphi_0)$ from the Fano metric.

Let S_k denote the cumulative metric for the first k channel symbols and suppose we have j_k errors among them. Then,

$$S_k = j_k s_e + (k - j_k) s_c. \quad (2.2.11)$$

Now we can more precisely state what we mean by "few" and "many" errors. Those error patterns for which S_k stays above the barrier at $-f$ contain "few" errors, and those error patterns for which the cumulative metric hits or crosses the barrier contain "many" errors. Few errors, i.e. $\min S_k > -f$, is equivalent to

$$j_k s_e + (k - j_k) s_c > -f, \text{ all } k \quad (2.2.12)$$

or

$$j_k < \frac{f}{s_c - s_e} + k \frac{s_c}{s_c - s_e} = r_k, \text{ all } k \quad (2.2.13)$$

In Figure 2.2.1 we illustrate inequality (2.2.13).

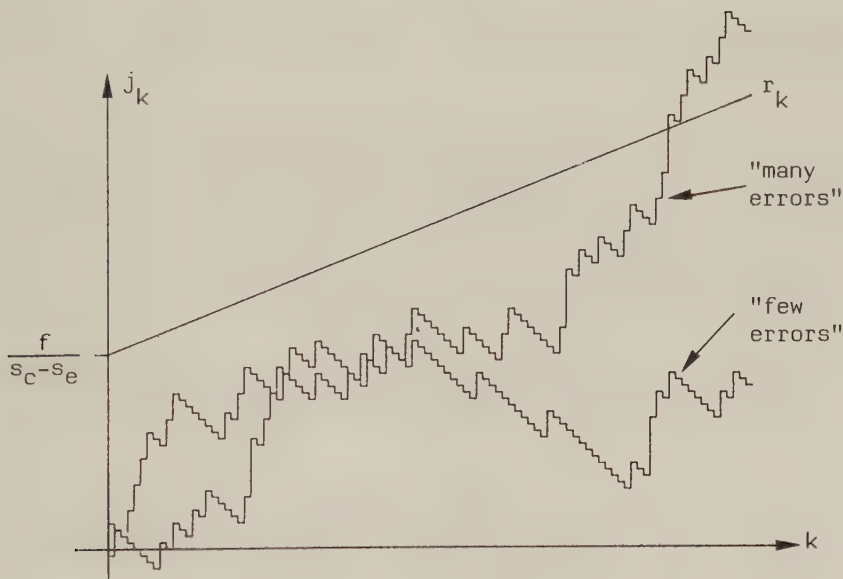


Figure 2.2.1 An illustration of the barrier at r_k which separates paths with "many" and "few" errors.

From inequality (2.2.6) we have

$$P[M] = P[\min_k S_k \leq -f] \leq 2^{-f}, \quad (2.2.14)$$

where f is a parameter to be chosen later.

2.3 FEW CHANNEL ERRORS - UNION BOUND

To upper bound the probability that we make an error when we decode the first information symbol and have few channel symbol errors, $P[E, F]$, we use the union bound and obtain

$$P[E, F] \leq \sum_k \sum_i P[E_{ki}, F], \quad (2.3.1)$$

where E_{ki} is the event that a path of length k and weight i is causing a decoding error.

A path has few channel errors only if it stays below the barrier in Figure 2.2.1 for all k . If we take all paths of length k with $j_k < r_k$ channel errors we will get all paths with "few" channel errors together with the paths with "many" channel errors that take one or more detours above the barrier. Hence, we have

$$P[E_{ki}, F] \leq P[E_{ki}, j_k < r_k], \quad (2.3.2)$$

where

$$P[E_{ki}, j_k < r_k] = \sum_{j_k < r_k} \binom{k}{j_k} p^{j_k} q^{k-j_k} P[E_{ki} | j_k]. \quad (2.3.3)$$

If we multiply the sum in equation (2.3.3) by $\eta^{-(r_k - j_k)}$, $0 < \eta \leq 1$, we obtain the inequality

$$P[E_{ki}, j_k < r_k] \leq \eta^{-r_k} \sum_{j_k < r_k} \binom{k}{j_k} (\eta p)^{j_k} q^{k-j_k} P[E_{ki} | j_k]. \quad (2.3.4)$$

Let us introduce

$$p_0 \triangleq \frac{\eta p}{\eta p + q} \leq p \quad (2.3.5)$$

and

$$q_0 \hat{=} 1 - p_0 = \frac{q}{\eta p + q} \geq q. \quad (2.3.6)$$

Substituting (2.3.5) and (2.3.6) into (2.3.4) and rearranging (2.3.4) we get

$$P[E_{ki}, j_k < r_k] \leq \left(\frac{pq_0}{p_0q}\right)^{r_k} \left(\frac{q}{q_0}\right)^k \sum_{j_k < r_k} \binom{k}{j_k} p_0^{j_k} q_0^{k-j_k} P[E_{ki} | j_k]. \quad (2.3.7)$$

Overbounding by summing over all $0 \leq j_k \leq k$, we have

$$P[E_{ki}, j_k < r_k] \leq \left(\frac{pq_0}{p_0q}\right)^{r_k} \left(\frac{q}{q_0}\right)^k P[E_{ki}]_{p_0}, \quad (2.3.8)$$

where $P[E_{ki}]_{p_0}$ is the probability that a decoding error is caused by a path of length k and weight i on an "improved" BSC with cross-over probability p_0 .

Using the Bhattacharyya bound (Viterbi et al, 1979) we obtain from (2.3.8)

$$P[E_{ki}, j_k < r_k] \leq \left(\frac{pq_0}{p_0q}\right)^{r_k} \left(\frac{q}{q_0}\right)^k (2\sqrt{p_0q_0})^i. \quad (2.3.9)$$

Using the definition of r_k given in (2.2.13), and rearranging (2.3.9) we get

$$P[E_{ki}, j_k < r_k] \leq \left(\frac{pq_0}{p_0q}\right)^{\frac{f}{s_c - s_e}} L^{k_D} i, \quad (2.3.10)$$

where

$$L = \frac{q}{q_0} \left(\frac{pq_0}{p_0q}\right)^{\frac{s_c}{s_c - s_e}} \quad (2.3.11)$$

and

$$D = 2\sqrt{p_0 q_0}. \quad (2.3.12)$$

Finally, we combine (2.3.1) and (2.3.2) with (2.3.10) and obtain the following upper bound on the probability of having few channel symbol errors and making an error when decoding the first information symbol:

$$\begin{aligned} P[E, F] &\leq \left(\frac{pq_0}{p_0q}\right)^{\frac{f}{s_c - s_e}} \sum_k \sum_i a_{ki} L^{kD^i} \\ &= \left(\frac{pq_0}{p_0q}\right)^{\frac{f}{s_c - s_e}} T(D, L), \end{aligned} \quad (2.3.13)$$

where a_{ki} is the number of paths of length k and weight i stemming from the root node. Thus, $T(D, L)$ is the generating function for the output sequence weights and lengths.

2.4 A TIGHTER VITERBI BOUND

We now show how to combine the bounds (2.1.1), (2.2.14) and (2.3.13) to obtain a new upper bound on the first-event error probability:

$$P[E] \leq \left(\frac{pq_0}{p_0q}\right)^{\frac{f}{s_c - s_e}} T(D, L) + 2^{-f}. \quad (2.4.1)$$

The bound (2.4.1) is valid for all f . By taking the derivative of the right-hand side of (2.4.1) we find that its minimum is obtained for

$$f_0 = \frac{(s_c - s_e)(\log(s_c - s_e) - \log T(D, L) - \log \log \frac{pq_0}{p_0q})}{\log \frac{pq_0}{p_0q} + s_c - s_e}. \quad (2.4.2)$$

Inserting (2.4.2) and rearranging (2.4.1) give the following upper bound:

$$P[E] \leq 2^{h(\gamma)} T(D, L)^\gamma, \quad (2.4.3)$$

where $h(\gamma)$ is the binary entropy function and

$$\gamma^{-1} = 1 + \frac{\log \frac{pq_0}{p_0q}}{s_c - s_e}. \quad (2.4.4)$$

Finally, we use (4) and (5) and obtain a new Viterbi type bound:

$$P[E] \leq \inf_{0 < p_0 \leq p} \inf_{p < a < 1} 2^{h(\gamma)} T(D, L)^\gamma, \quad (2.4.5)$$

where

$$\gamma^{-1} = 1 + \frac{\log \frac{pq_0}{p_0q}}{\log \frac{qa}{p(1-a)}}, \quad (2.4.6)$$

$$D = 2\sqrt{p_0q_0} \quad (2.4.7)$$

and

$$L = \frac{q}{q_0} \left(\frac{pq_0}{p_0q} \right)^{\frac{\log \frac{q}{1-a}}{\log \frac{qa}{p(1-a)}}} \quad (2.4.8)$$

Our bound is significantly better than Viterbi's bound for rates $R_{\text{comp}} < R < C$. His bound can be obtained as a special case of (2.4.5) by choosing $p_0 = p$.

2.5 A TIGHTER Van de MEEBERG TYPE BOUND

Let P_i be the error probability for two binary codewords at distance i (Viterbi, 1971). Van de Meeberg (Meeberg, 1974) used the fact that $P_{2i} = P_{2i-1}$ to tighten the Bhattacharyya bound and showed that

$$P_{2i} \leq \left(2^{\delta} - 1\right) 2^{-2\delta} (2\sqrt{p})^{2i}, \quad (2.5.1)$$

where

$$\delta = \frac{d_{\infty} + 1}{2}. \quad (2.5.2)$$

Van de Meeberg used the inequality

$$\frac{q^i - p^i}{q - p} \leq 1. \quad (2.5.3)$$

We notice that

$$\frac{q^i - p^i}{q - p} = q^i \frac{1 - (p/q)^i}{1 - 2p} < (\sqrt{q})^{2i} \frac{1}{1 - 2p}. \quad (2.5.4)$$

For $p \leq 0.38$ and $d_{\infty} > 4$ (see Figure 2.5.1) the bound (2.5.4) is tighter than (2.5.3), and we have the following (for all practical values of p and d_{∞} slightly improved) Van de Meeberg type bound

$$P_{2i} < \left(2^{\delta} - 1\right) \frac{2^{-2\delta}}{1 - 2p} (2\sqrt{pq})^{2i}, \quad (2.5.5)$$

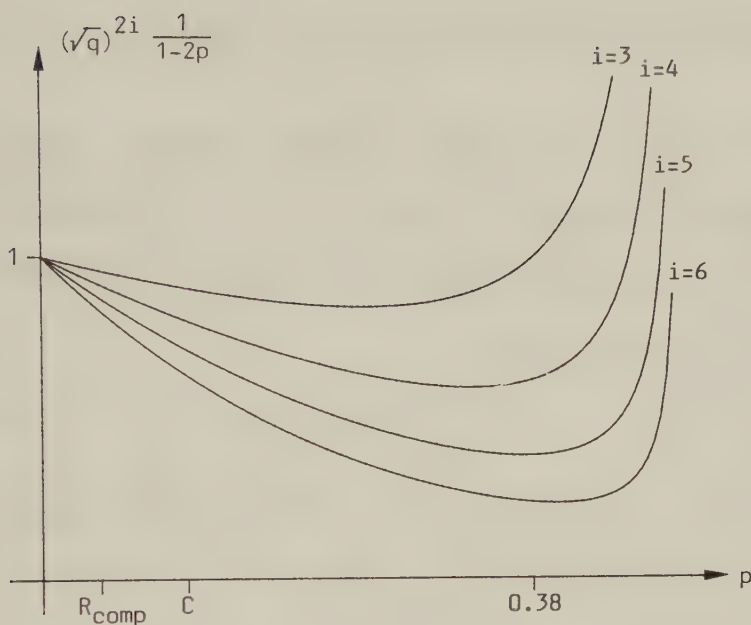


Figure 2.5.1 Factor of Van de Meeberg's bound.

Since $P_{2i} = P_{2i-1}$, $i \geq 1$, we can now rewrite our Viterbi type bound (2.4.5) as

$$P[E] < \inf_{0 < p_0 \leq p} \inf_{p < a < 1} 2^{h(\gamma)} \left[\left(\frac{2^\delta - 1}{\delta} \right) \frac{2^{-2\delta}}{1 - 2p_0} M(D, L) \right]^\gamma \quad (2.5.6)$$

where

$$M(D, L) = [T(D, L) + T(-D, L)]/2 + D[T(D, L) - T(-D, L)]/2, \quad (2.5.7)$$

and γ , δ , D and L are given in (2.4.6), (2.5.2), (2.4.7) and (2.4.8), respectively.

2.6 GENERATING FUNCTION $T(D,L)$

Let $i_0, i_1, i_2, \dots$ be the information sequence, then $s_u = [i_{u-1}, i_{u-2}, \dots, i_{u-M}]$ is the state of a rate 1/2 convolutional encoder of memory M . The state $s_{u+1} = [i_u, i_{u-1}, \dots, i_{u-M+1}]$ is stemming from the states $s_u = [i_{u-1}, \dots, i_{u-M+1}, 0]$ and $s_u = [i_{u-1}, \dots, i_{u-M+1}, 1]$:

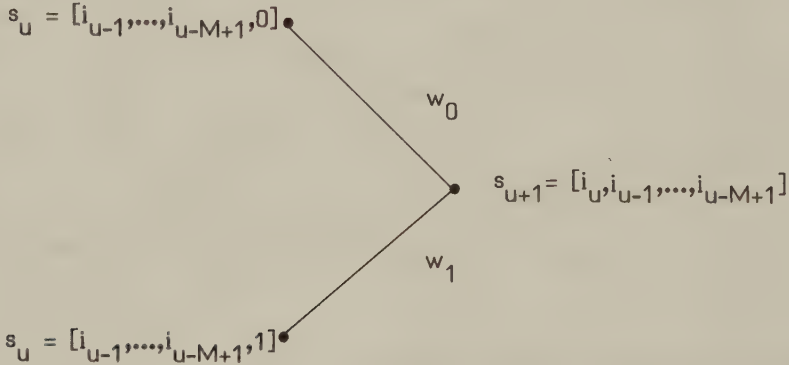


Figure 2.6.1 Possible "fathers" of a state.

The branch weights w_0 and w_1 are given by

$$w_0 = \sum_{k=1}^2 \sum_{j=0}^{m-1} i_{u-j} g_j^{(k)} \quad (2.6.1)$$

and

$$w_1 = \sum_{k=1}^2 \left(\sum_{j=0}^{m-1} i_{u-j} g_j^{(k)} \oplus g_M^{(k)} \right), \quad (2.6.2)$$

where $G^{(k)} = [g_0^{(k)}, g_1^{(k)}, \dots, g_M^{(k)}]$ for $k=1,2$ are the code generators, and the sum over j is evaluated in $GF(2)$.

Let $\xi[x, y, \dots, z]$ be a dummy variable for the partial paths to state $[x, y, \dots, z]$ in the state diagram (Viterbi et al, 1979). Then by definition we have

$$\xi[x, y, \dots, z] = D^w L^2 \xi[y, \dots, z, 0] + D^w L^2 \xi[y, \dots, z, 1], \quad (2.6.3)$$

where the D and L factors are used to determine the weight and length (in channel symbols) of a given path, respectively. The variable $\xi[0,0,\dots,0] = 1$ and the generating function for the output sequences is given by

$$T(D, L) = D^w L^2 \xi[0, \dots, 0, 1]. \quad (2.6.4)$$

The desired generating function $T(D, L)$ is a polynomial in D and L and for relatively short codes it is easily determined by solving the linear equations (2.6.3) for all states. This is equivalent to find an eigenvector to a sparse matrix. However, already for codes of moderate memory, say $M = 10$, this is not feasible. Therefore, we use the actual numerical values of D and L obtained from equations (2.4.7) and (2.4.8) and solve the corresponding numerical system of linear equations by iteration. We use $\xi[0,0,\dots,0] = 1$, $\xi[0,0,\dots,1] = \dots = \xi[1,1,\dots,1] = 0$ as a start vector and iterate the $2^M - 1$ equations in the following order: $\xi[1,0,0,\dots,0]$, $\xi[0,1,0,\dots,0]$, $\xi[1,1,0,\dots,0]$, . . ., $\xi[1,1,1,\dots,1]$, $\xi[0,1,1,\dots,1]$, $\xi[1,0,1,\dots,1]$, . . ., $\xi[0,0,0,\dots,1]$. If the most recently calculated estimates of ξ are used in the iterations we can quickly determine $\xi[0,0,\dots,1]$ and hence $T(D, L)$.

2.7 NUMERICAL RESULTS

We calculate our Viterbi type bound (2.4.5) for the standard rate $1/2$ convolutional code, viz. the memory $M = 2$ code with code generators $G^{(1)} = 5$, $G^{(2)} = 7$ (octal notation). In Figure 2.7.1 we compare our bound both with simulations and with Viterbi's bound:

$$P[E] < T(D, L) |_{D = 2\sqrt{pq}, L = 1} \quad (2.7.1)$$

Our bound is significantly better than Viterbi's for rates above R_{comp} .

For the same code we compare (Figure 2.7.2) our Van de Meeberg type bound (2.5.6) with simulations and with both Van de Meeberg's and Post's bounds. Our bound is similar to Post's bound but significantly better than Van de Meeberg's bound. For channels with small cross over probability our bound is of course equivalent to Van de Meeberg's bound.

For a slightly longer code, viz. the memory $M = 4$ code with generators $G^{(1)} = 72$ and $G^{(2)} = 62$ ($d_{\infty} = 7$), our bound is significantly better not only than Van de Meeberg's bound but also than Post's bound (Figure 2.7.3).

We close this chapter by showing the dramatic improvement of Van de Meeberg's bound obtained for rates between R_{comp} and channel capacity, C , for even longer codes, viz. the two ODP codes (Johannesson, 1975) of memory $M = 8$ ($d_{\infty} = 12$) and $M = 15$ ($d_{\infty} = 18$). The curves are given in Figure 2.7.4 and 2.7.5.

While our new bound is rather quickly evaluated for short codes, approximately one minute on a VAX-11/780 for our $M = 4$ curve, it is very time consuming to obtain the bound for long codes, approximately nine hours for our $M = 15$ curve.

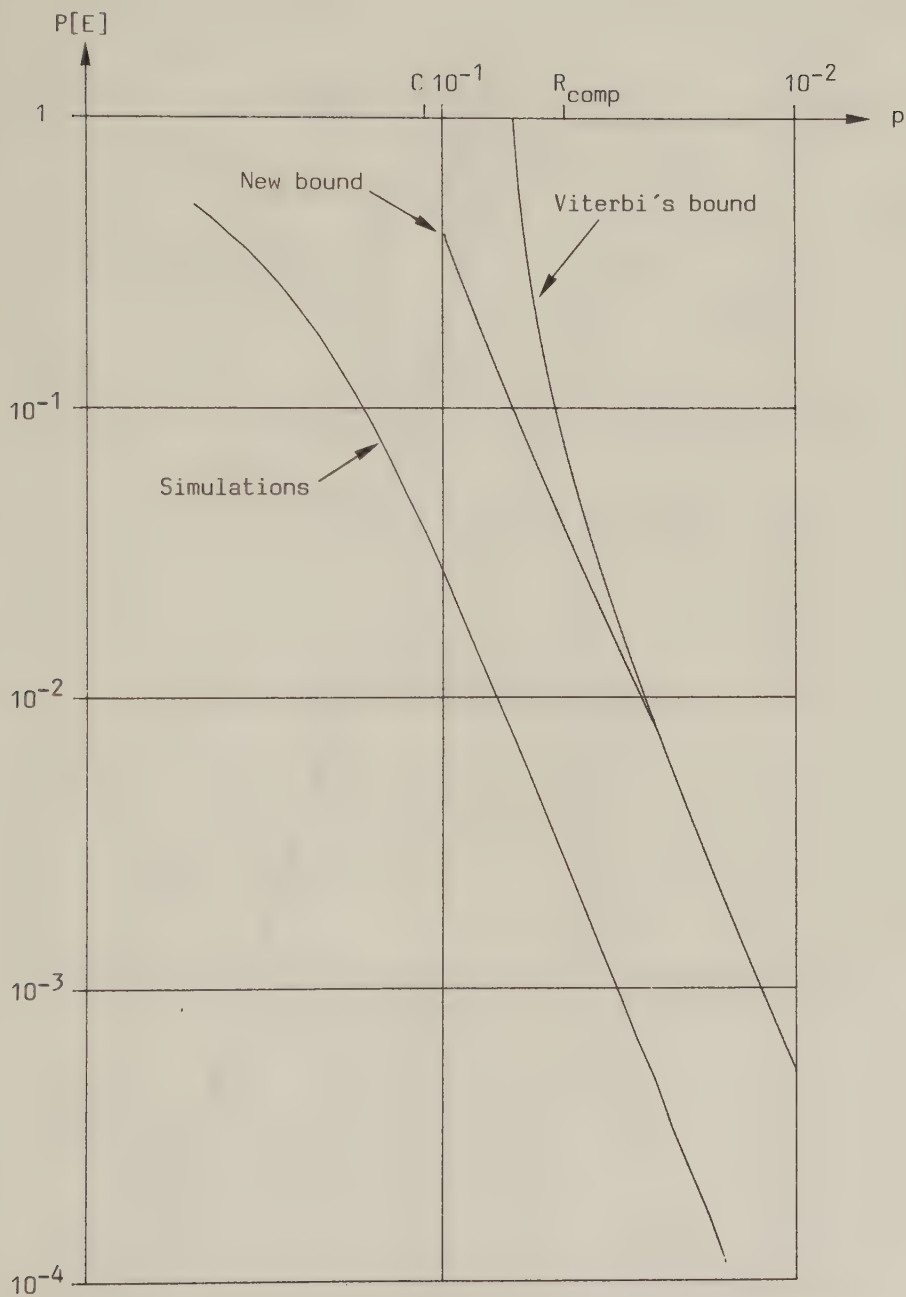


Figure 2.7.1 New Viterbi type bound compared with Viterbi's bound and simulations for the $R = 1/2$, $M = 2$, ODP code with $G^{(1)} = 5$ and $G^{(2)} = 7$.

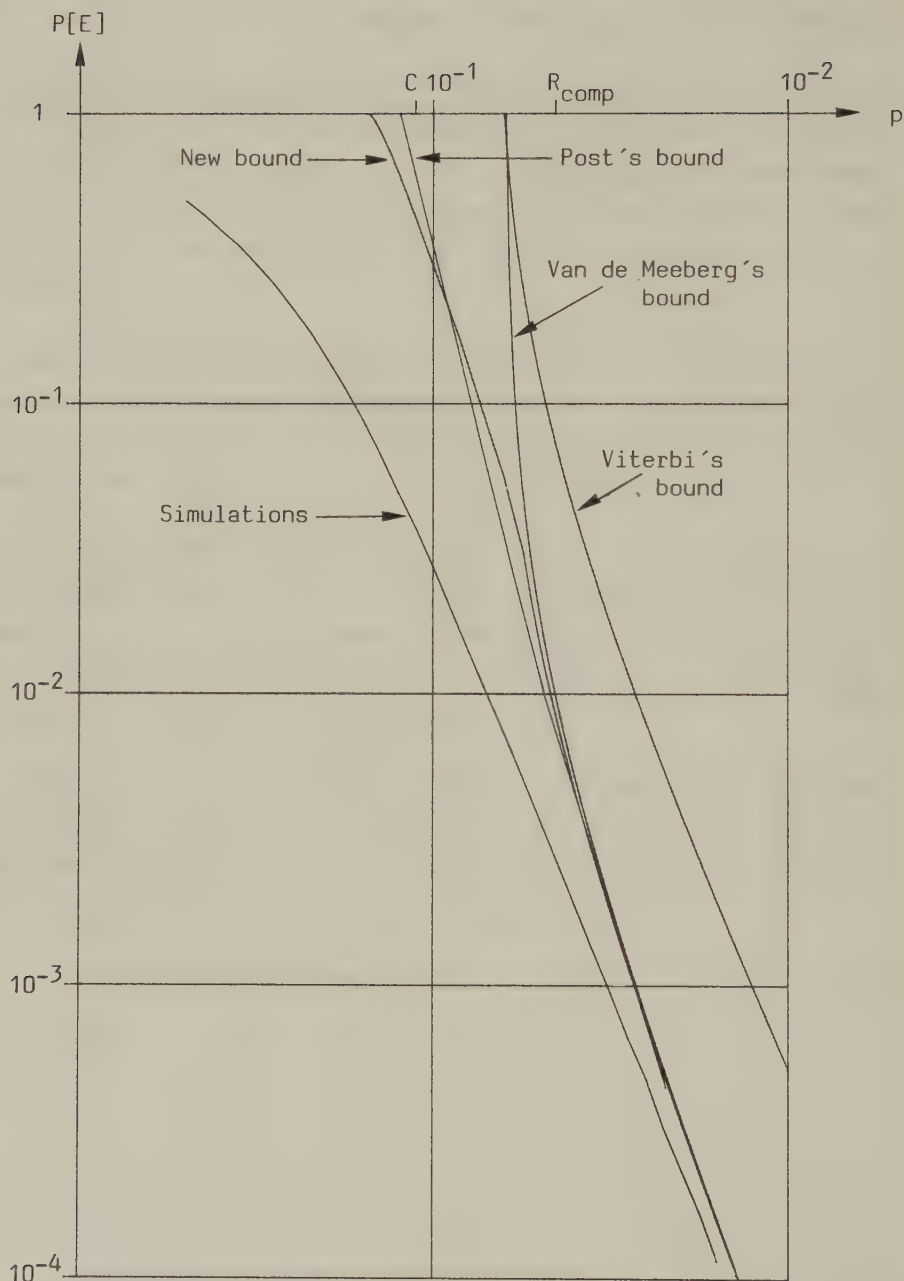


Figure 2.7.2 New Van de Meeberg type bound compared with Van de Meeberg's and Post's bounds and with simulations for the $R = 1/2$, $M = 2$, ODP code with $G^{(1)} = 5$ and $G^{(2)} = 7$.

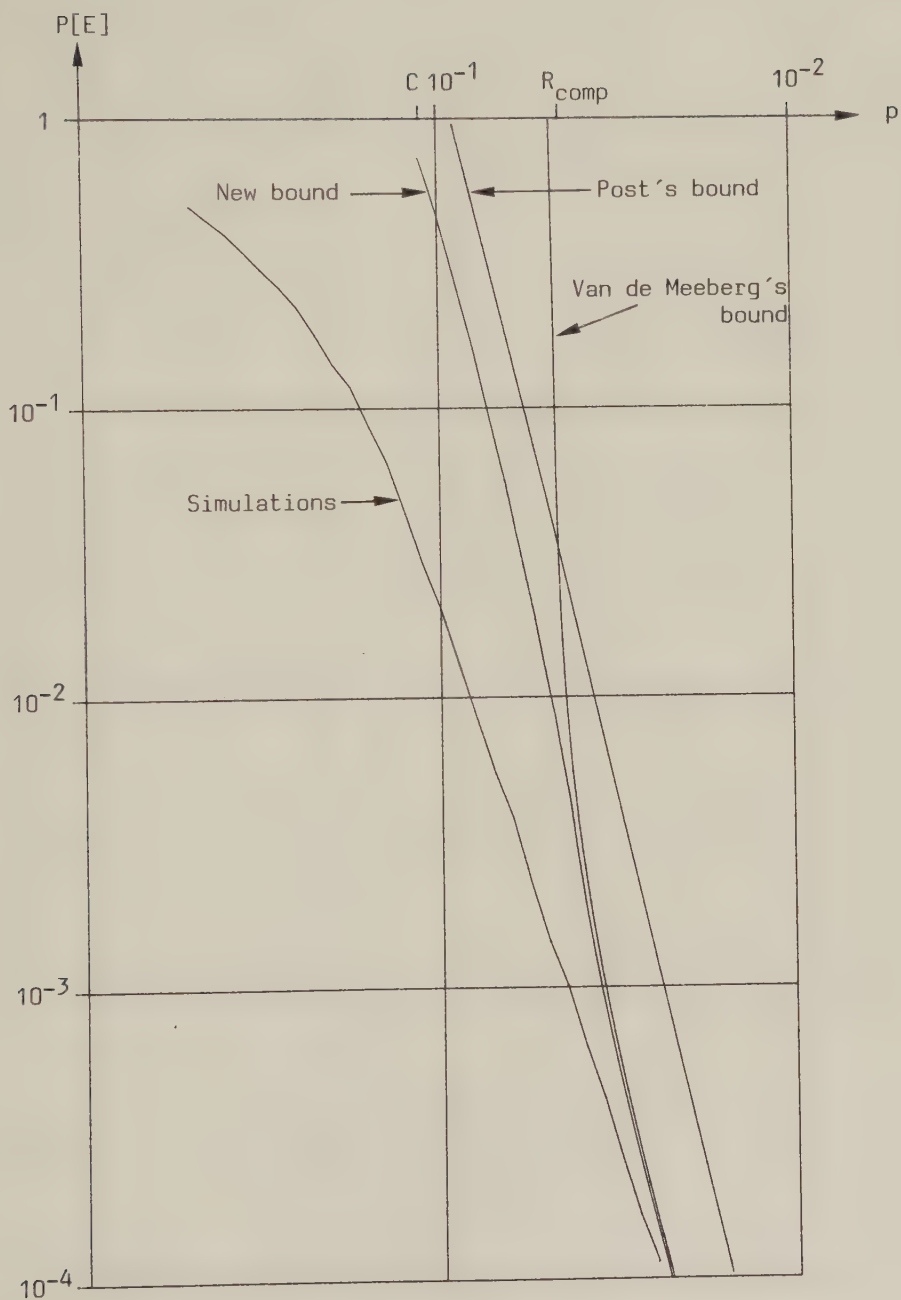


Figure 2.7.3 New Van de Meeberg type bound compared with Van de Meeberg's and Post's bounds and with simulations for the $R = 1/2$, $M = 4$, ODP code with $G^{(1)} = 72$ and $G^{(2)} = 62$.

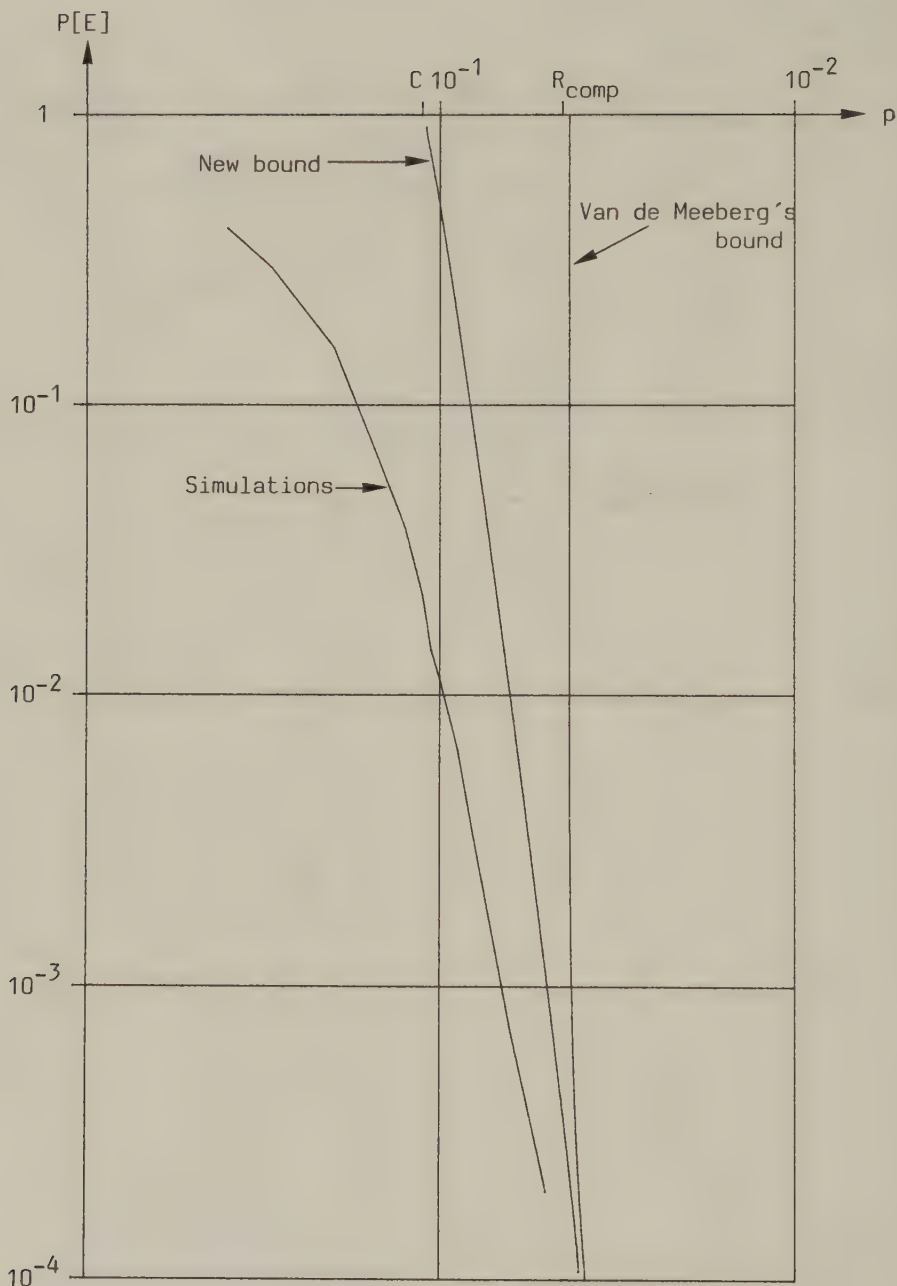


Figure 2.7.4 New Van de Meeberg type bound compared with Van de Meeberg's and simulations for the $R = 1/2$, $M = 8$, ODP code with $G^{(1)} = 557$ and $G^{(2)} = 751$.

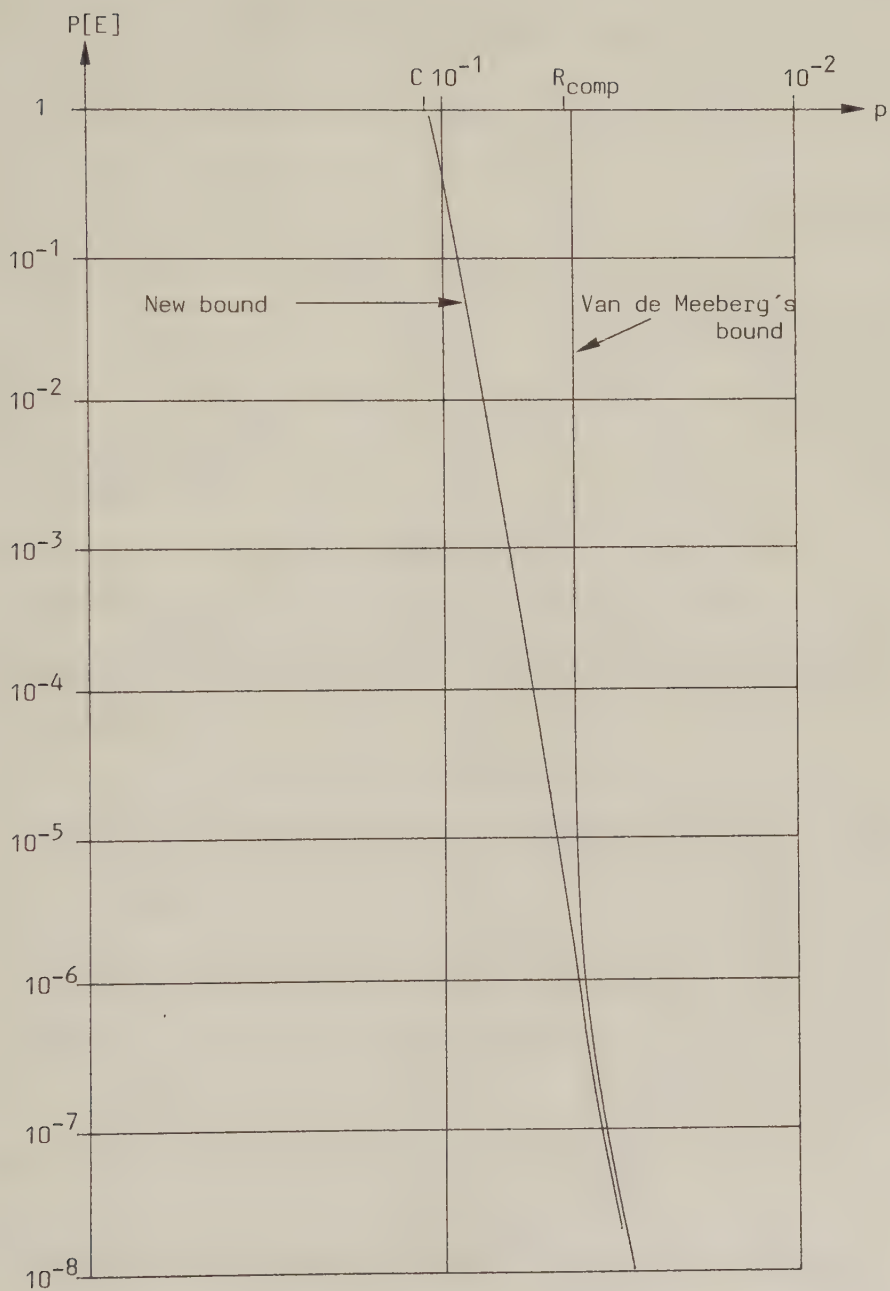


Figure 2.7.5 New Van de Meeberg type bound compared with Van de Meeberg's for the $R = 1/2$, $M = 15$, ODP code with $G^{(1)} = 467724$ and $G^{(2)} = 621134$.

An erasure.

COMPUTING DISTANCE SPECTRUM

3.1 Problem

Over the past decade there has been a significant increase in using sequential decoding to achieve reliable communication. We can expect that the demand for communication with extremely low error probability will continue to grow.

It is well-known (Viterbi et al, 1979) that the distance spectrum, i.e. the list of the number of paths of weights $d_{\infty}, d_{\infty}+1, d_{\infty}+2, \dots$ which depart from the all zero path at the root node in the tree and do not reach the zero state until their termini, is the principal determiner of decoding error probability for a convolutional code. It has also been observed [(Massey et al, 1971), (Chevillat et al, 1976), (Johannesson, 1975)] that an optimum distance profile is desirable to obtain a good computational performance with sequential decoding. Thus, it is important to find methods for constructing convolutional codes with both a good distance spectrum and a good distance profile.

So far there has been little success in finding very good convolutional codes by algebraic methods. In 1965 Julian J. Bussgang wrote (Bussgang, 1965):

"The problem of systematically constructing a binary convolutional code capable of correcting the largest number of errors within a given constraint length is as yet unsolved."

This is still true. Most codes that are used in practice have been found by computer search.

Since the j -th order column distance, d_j , is evaluated by a search in the code tree only to depth $j+1$, it is a "relatively simple" task to determine the distance profile, which is defined as the $(M+1)$ -tuple $\mathbf{d} = [d_0, d_1, \dots, d_M]$, where M is the memory of the code. On the other hand a direct evaluation of the spectrum needs a search among unreasonably many paths and becomes practically impossible except for rather small memories.

Most algorithms for finding distances presented in the literature [(Bahl et al, 1968), (Divsalar, 1978), (Larsen, 1973), (Odenwalder, 1970)] suffer from the condition that either being slow or using too much computer memory. In the latter group are those algorithms that invert the transition matrix. The rank of the code transition matrix grows exponentially with the code memory. In this chapter we will present an algorithm which is fast and uses virtually no computer memory.

All of the following algorithms assume that the code is **non catastrophic**, i.e. that $\gcd(G^{(1)}(D), G^{(2)}(D)) = 1$ (Massey et al, 1968).

3.2 A basic algorithm

One way to save computer memory is to use tree algorithms. In this section we introduce **tree algorithms** for computing distance parameters of convolutional codes.

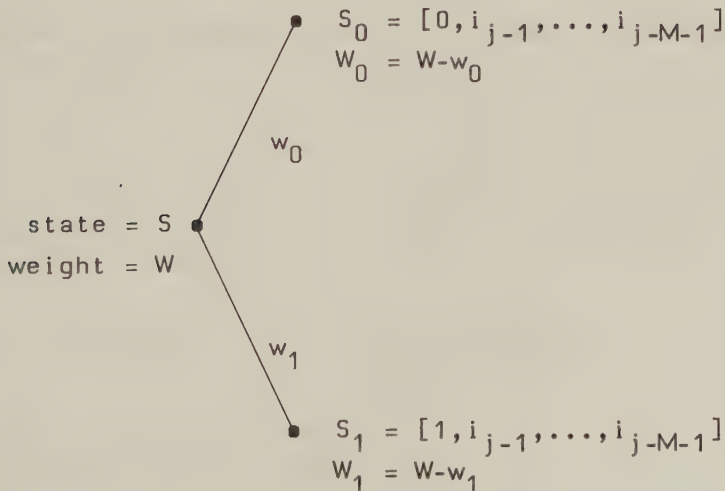
To compute the number of paths with a given distance to the all zero path we count the number of sequences with a weight equal to

this distance and stemming from the zero state and terminating for the first time at the zero state.

Suppose we are in an arbitrary **node** in the tree and that we have produced channel symbols whose total weight is W_t . This means that in each subtree stemming from this node we have to "spend" the weight distance- W_t . Hence, let us label each node with the **state** and **remaining weight**, i.e. $W = \text{distance} - W_t$.

Let $i_0, i_1, i_2, \dots$ be the information sequence. Then $S = [s_1, s_2, \dots, s_M]$, where $s_n = i_{j-n}$ for $n=1, 2, \dots, M$ and $i_k = 0$ for $k < 0$, is the **state** of the rate 1/2 convolutional encoder with memory M , and s_n is called a **state variable**. From the state we will have two successor states, S_0 and S_1 , corresponding to information symbols i_j equal to zero and one, respectively.

For given code generators we can of course use the state of a node to determine the weights, w_0 and w_1 of the branches stemming from that node. By using these weights together with the node weight W we can determine the two new node weights W_0 and W_1 . See Figure 3.2.1.



$$w_{ij} = \sum_{k=1}^2 \sum_{n=0}^M i_{j-n} g_n^{(k)},$$

Figure 3.2.1 Successor nodes at time j , where $G^{(k)}$ for $k=1,2$ are the code generators, and the sum over n is taken modulo 2.

We are able to explore a subtree if and only if the new node weight, w_{ij} is nonnegative and if the state of the new node S_{ij} differs from the zero state.

Let us arbitrarily give **priority** to the zero branch when we have to select between two new possible nodes.

If due to the restrictions above we can not move **forward** in the tree we will move **backwards**. To do this we have to remember all of the previous information symbols so that we can move backwards until we find a new "one" branch with a non-negative node weight. A **stop condition** appears when we reach the root. While moving backwards we use the notations $w_{-i_{j-1}}$ and $S_{-i_{j-1}}$ for the weight and the state respectively, that led us to our present state.

A straight-forward algorithm for this tree search is given in Figure 3.2.2.

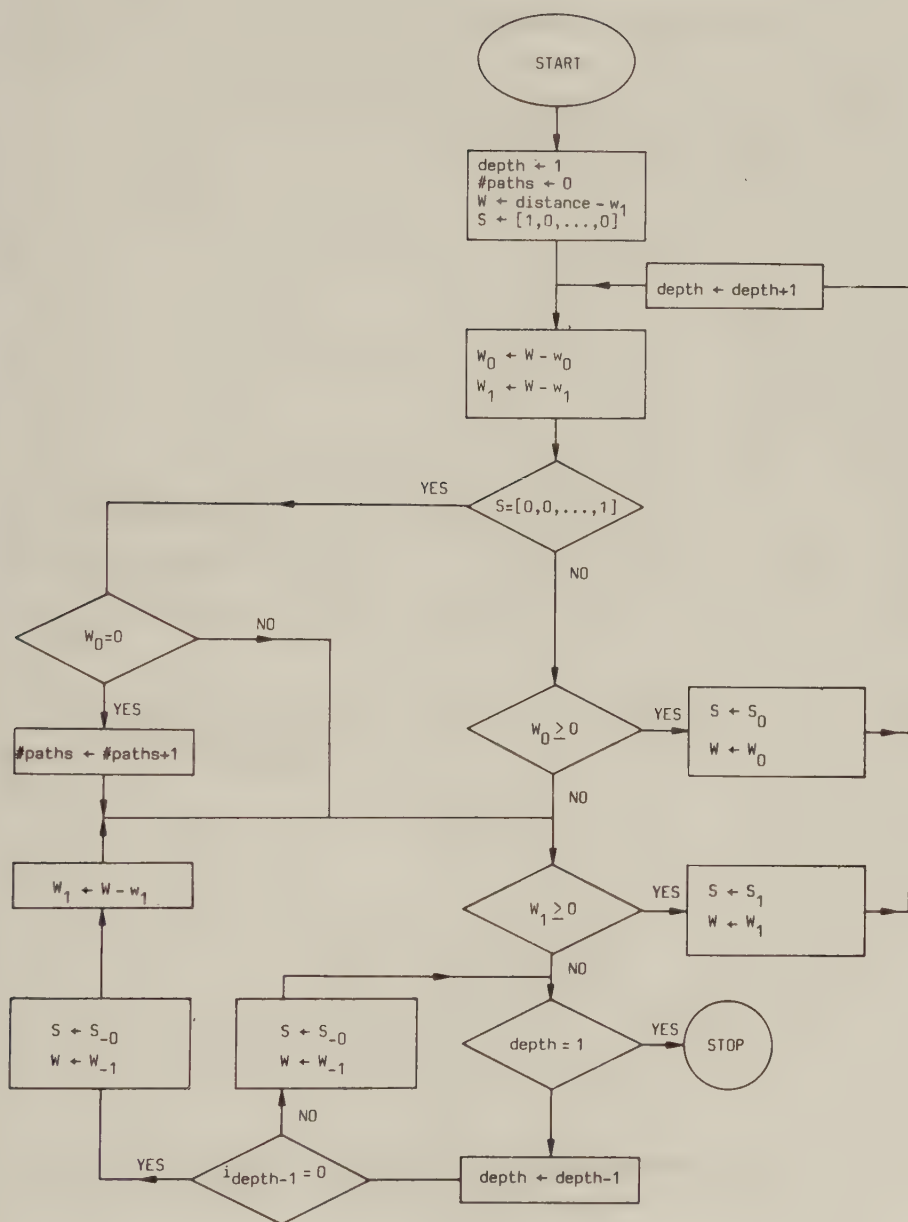


Figure 3.2.2 A BASIC algorithm, BA, for spectrum parameters.

To measure the performance of a tree algorithm we use the total

number of nodes visited. Each time we visit a node, regardless if we have been there before or not, we increase this number. As an example, let us use the code $G = (74, 54)$ given in Figure 1.2.2. The explored tree is shown in Figure 3.2.3

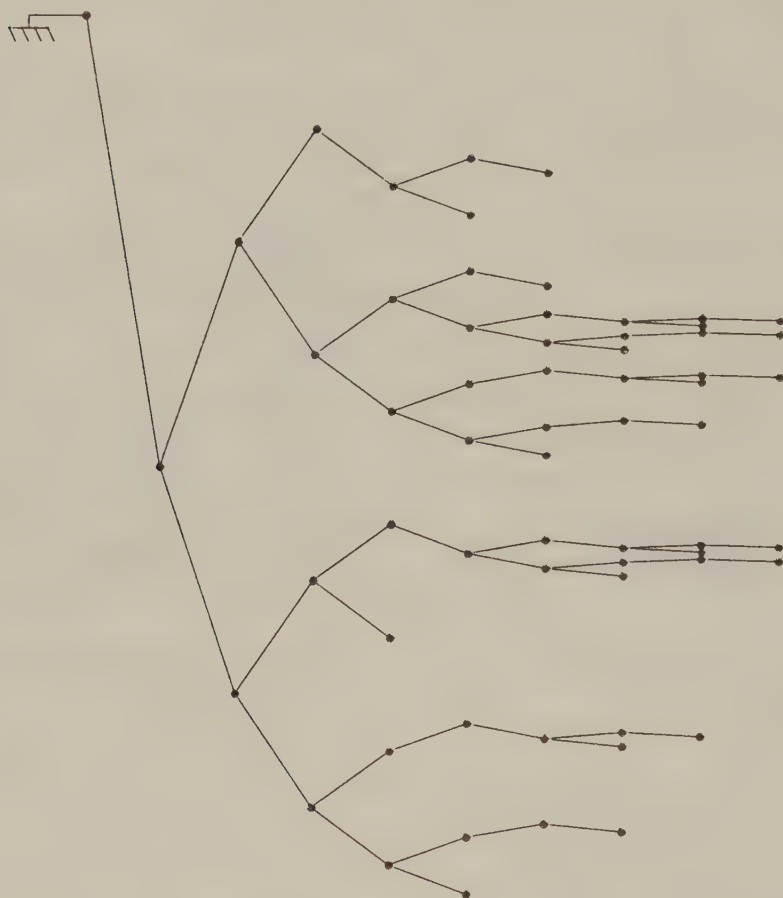


Figure 3.2.3 The tree searched using BA and $G = (74, 54)$. In this case we visit 121 nodes.

Generally speaking, the number grows exponentially with the distance. But this should not prevent us from determining the distance parameters for larger distances.

3.3 An improved basic algorithm

The basic algorithm uses a threshold of zero when it compares each new node weight. Is this the optimum choice of the threshold? No, actually this threshold is our worst choice. The range of the threshold can easily be determined by making the following observation:

The only way to reach the all zero state is to branch from the state $[0,0,\dots,1]$. As an example we can see that for the code in Figure 1.2.2, each generator $G^{(k)}$ is "connected to" s_M , the state variable containing i_{j-M} . It means that in this case, w_0 is equal to two. Generally w_0 will be the sum of $g_M^{(1)}$ and $g_M^{(2)}$. Hence, when leaving the tree we must have a weight equal to this sum! We could use this sum as our threshold. Notice that the weight can not fall below this threshold anywhere in the tree, or else we will not be able to leave the tree.

The difference between this improved basic algorithm, IBA, and BA can be seen in Figure 3.3.1.

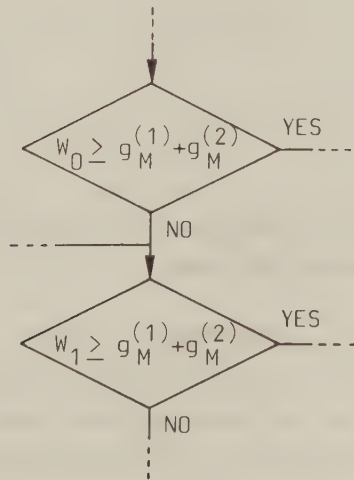


Figure 3.3.1 Improved Basic Algorithm.

It is also possible to give theoretical support to this observation:

Assume that the output, the channel symbols, from the encoder are equally likely. Moreover, the output corresponding to a one on the input must be the complement of the output produced with a zero as input. This is always the case when $d_0 = 2$. Assume that we have the tree produced by IBA. Then every leaf has a weight equal to two. Consider a path in the BA tree from a node, which corresponds to a leaf in the IBA tree, leading to a leaf. Such a path has a weight equal to two. If its length is x , then it is one of $\binom{2x}{2} = x(2x-1)$ paths. Each of these paths has the probability 2^{-2x} . Since a branch from a leaf has a weight equal to one, the probability for a node with weight equal to zero being a leaf is $2/4$.

Since the length must be less than some value X , we have that the mean value of these lengths, $E[x]$, is given by (3.3.1).

$$E[x] = \sum_{x=1}^X x \cdot \frac{2}{4} \cdot x(2x-1) 2^{-2x} \quad (3.3.1)$$

If the above stated is valid for the whole tree, then we get the quotient of performance, QP , between the BA and IBA as

$$QP = 2^{-E[x]}. \quad (3.3.2)$$

The agreement between this and the measured values is very tight. QP as a function of X can be seen in Table 3.3.1.

X	$E[x]$	QP
1	0.25	0.84
2	1.00	0.50
3	1.70	0.30
4	2.14	0.22
5	2.36	0.19
6	2.45	0.18
7	2.49	0.17
8	2.51	0.17
9	2.51	0.17
10	2.51	0.17

Table 3.3.1 QP as function of X.

Does the code contain more information that can reduce the computation time? The above improvement was only based upon the observation of how the code is connected at the end, so never say die.

3.4 Applying a stack

When we use one of the above algorithms (actually it is one algorithm with different thresholds) we might notice that we calculate the node weight for a specific node twice. Once when we move forward and once again while moving backwards. Is this necessary? No, we should be able to gain a factor of two!

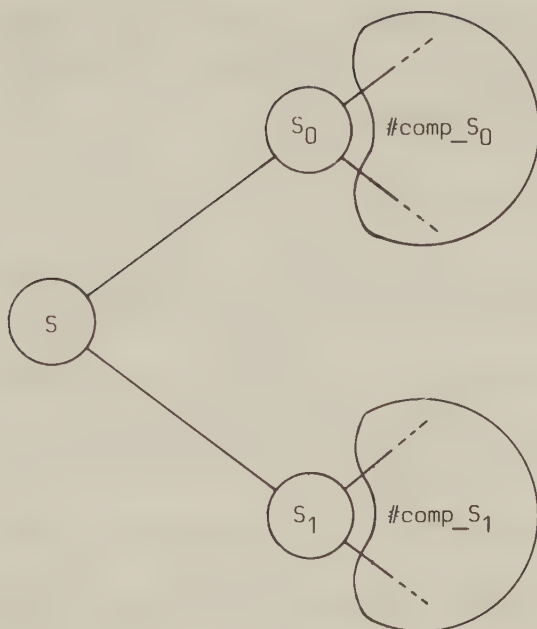


Figure 3.4.1 # computations in subtrees.

This is true because of the following:

Assume that we use IBA and are in the state S looking forward. If it is known how much time we have to spend in each subtree, denoted $\#comp_{S_0}$ and $\#comp_{S_1}$, stemming from the next states S_0 and S_1 , then we also have the time for the subtree stemming from S . The time is given by 3.4.1.

$$\#comp_S \doteq \#comp_{S_0} + \#comp_{S_1} + 4 \quad (3.4.1)$$

The number 4 arises from the fact that the algorithm searches the tree according to

$$S \rightarrow S_0 \rightarrow subtree_0 \rightarrow S \rightarrow S_1 \rightarrow subtree_1 \rightarrow S. \quad (3.4.2)$$

Let us introduce the following strategy:

While both $node_0$ and $node_1$ are achievable, we will save the

node₁ on the **stack** . Hence if we cannot move forward, we simply pop a node from the stack, if the stack is not **empty** . So, if we use this, (3.4.2) will change to

$$S \rightarrow S_0 \rightarrow \text{subtree}_0 \rightarrow S_1 \rightarrow \text{subtree}_1, \quad (3.4.3)$$

and we only have to add the number 2 in (3.4.1). In this latter case $\# \text{comp}_{S_k}$ will differ, go halves, from those used in (3.4.1).

If one of the next states in Figure 3.4.1 is not reachable, then the number 4 in (3.4.1) should be 2, and consequently also in this case we will halve the number of computations if we use a stack.

Conclusion: Independent of the tree algorithm, a stack used in this manner will halve the number of computations. Thus, if we use IBA together with a stack, the total time will have decreased to below 10% compared to BA.

3.5 A FAST algorithm

The code $G = (74, 54)$ used above has only one path with weight 6 ($=d_\infty$). This path corresponds to the information sequence 11000. We show this sequence in a **trellis** , Figure 3.5.1.

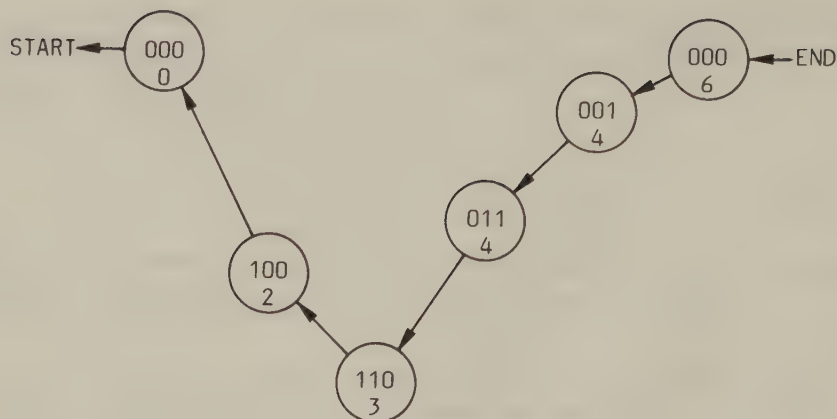


j:	0	1	2	3
d_j :	2	3	3	4

Figure 3.5.1 A part of the trellis for the code $G = (74,54)$ together with the distance profile.

Along our trip through this path, the distance profile tells us how much the decrease in each new node weight **at least** must have been, compared to our tested distance. So if we simply shift this profile one step to the right it tells us the actual minimum decrease to the actual node. Let us keep this shifted profile.

If we instead start at the end of the path, END, and move towards START, we get the same weight of the path (of course!), but each node will have a different weight. See Figure 3.5.2.



j:	0	1	2	3	(shifted)
d_j :	2	3	3	4	

Figure 3.5.2 Moving towards START in the trellis with the code $G = (74,54)$.

In this case we can use the shifted distance profile to **lower bound** the weight in each node. If the weight decreases below this limit, it will be impossible to reach the all zero state at the end (START) with a weight of zero. To use the distance profile as a lower bound works for every path from END to START.

Let us assume that we are in a state $S \neq [0, \dots, 0]$. We notice that the weight of a path stemming from this node, starting with a one branch and leading to a zero state, is lower bounded by $\tilde{d}_M$, where $\tilde{d}_M$ is the M-th order column distance of the **reversed code**, i.e. the code with generators

$$\tilde{G}^{(k)} = [g_M^{(k)}, g_{M-1}^{(k)}, \dots, g_0^{(k)}] \text{ for } k=1, 2. \quad (3.5.1)$$

From the node S we can reach the nodes S_0 and S_1 , where:

$$S = [0, \dots, 0, 1, ?, ?, \dots, ?]$$

n-1 zeros

$$S_0 = [0, 0, \dots, 0, 1, ?, \dots, ?]$$

n zeros

$$S_1 = [1, 0, \dots, 0, 1, ?, \dots, ?]$$

n-1 zeros

(3.5.2)

The minimum weights of paths stemming from the nodes S_0 and S_1 are lower bounded by $\tilde{d}_{M-n-1}$ and $\tilde{d}_{M-1}$, respectively, where $\tilde{d}_{M-n-1}$ and $\tilde{d}_{M-1}$ are the $(M-n-1)$ -th and $(M-1)$ -th order column distance for the reversed code, respectively.

We shall now describe a fast algorithm for computing the number of paths of a given distance in the subtree which corresponds to the first information symbol $i_0 = 1$. Starting at state $S_1 = [1, 0, \dots, 0]$ with weight $W = \text{distance} - w_0$ we reduce this weight by the weights of the branches that we pass when the code tree is searched for nodes with both weight and state equal to zero. From the distance profile of the reversed code and from the state of each explored node we know a lower bound of the weight of any path leading to a zero state. If the weight is less than this bound we will always reach zero weight at a nonzero state. Hence, it is only necessary to extend such a node if the node weight is larger than or equal to this bound.

Our algorithm is shown in Figure 3.5.3 and an example in Figure 3.5.4.

The analysis of a rate $1/N$, $N = 2, 3, \dots$, convolutional code starts with the calculation of the distance profile of both the code and the reversed code. If $d \geq \tilde{d}$, the reversed code generators, else the code generators, are plugged into this algorithm. The generalization of the algorithm to rates K/N is obvious.

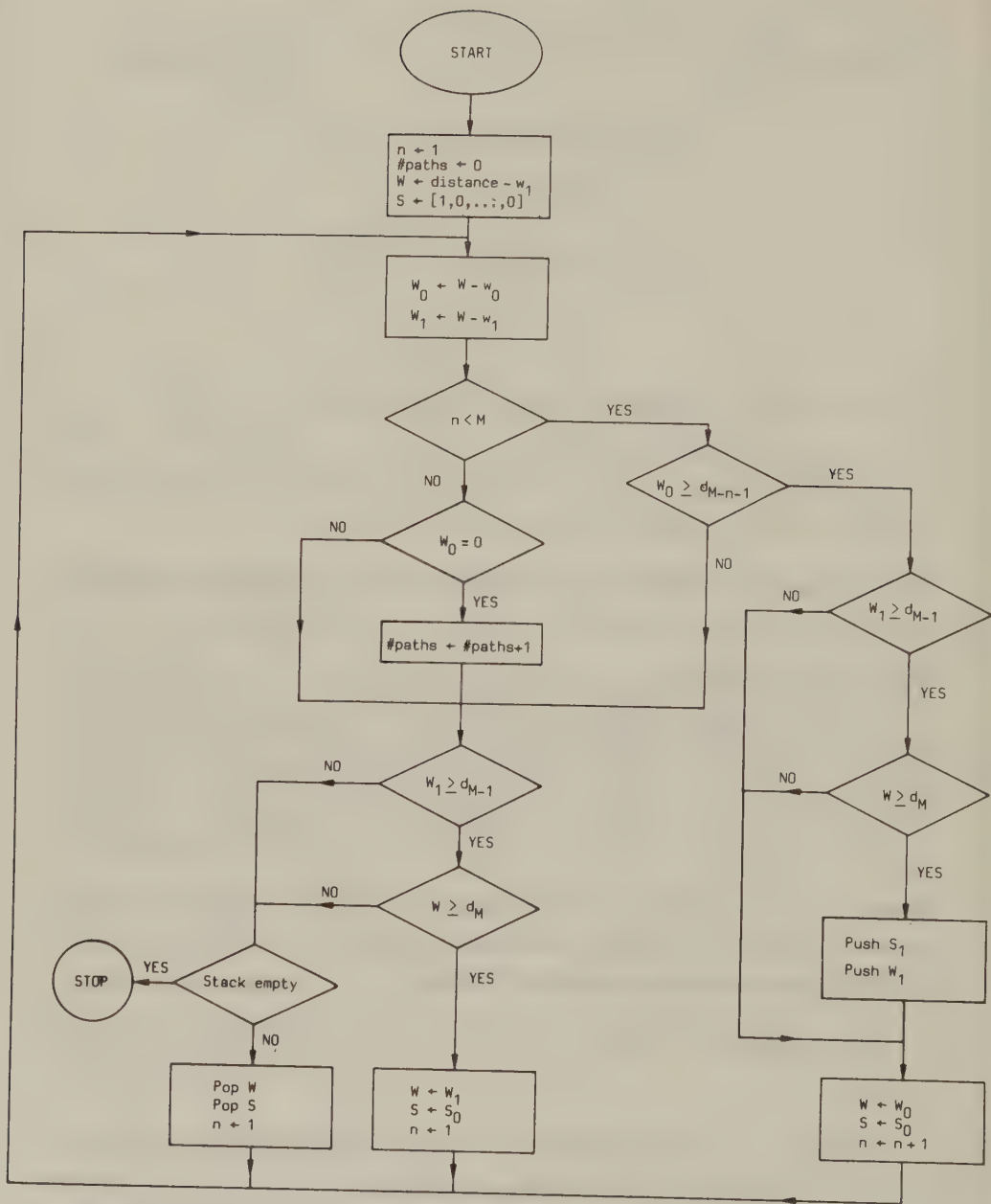


Figure 3.5.3 A FAST algorithm for computing distance spectrum parameters of convolutional codes with rate $R = 1/N$.



Figure 3.5.4 The tree searched using FAST and $G = (74,54)$. In this case we go through 5 nodes.

Since we are looking for convolutional codes with an optimum distance profile it is interesting to notice that **the better the distance profile is, the faster the algorithm runs!** This will easily be verified if we try to calculate spectrum parameters for systematic codes. The reverses of these codes have a constant distance profile ($=1$) up to $M-1$. In this case FAST corresponds to IBA, using a threshold equal to one. Figure 3.5.5 demonstrates the efficiency of FAST used on nonsystematic ODP codes.

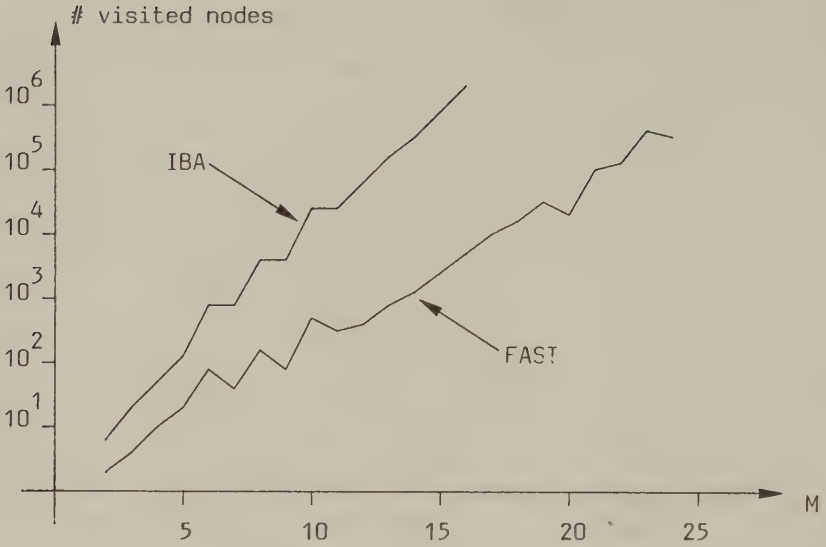


Figure 3.5.5 Comparison between FAST and IBA with ODP codes.

3.6 Modifications of the FAST algorithm

Some observations should be made on the FAST algorithm. If we are interested only in computing the number of paths with weight d_∞ , then we should apply a modification according to Figure 3.6.1. This will decrease the computation time if the free distance is unknown.

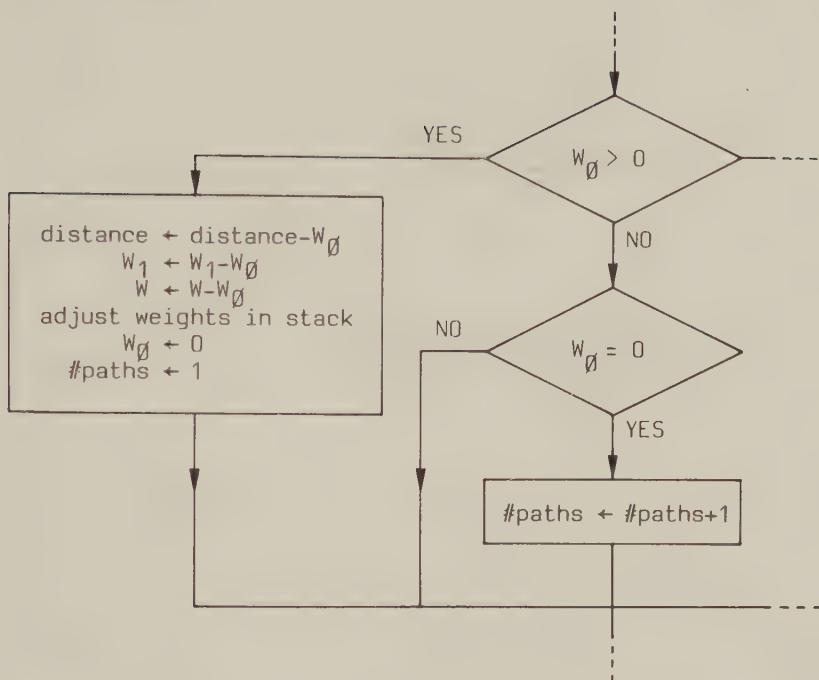


Figure 3.6.1 Modification of the FAST algorithm when computing free distance.

In Chapter 2 we saw that it is also desirable to know the **length**, L , of each path with tested distance. To use the FAST algorithm to receive this information, we must incorporate a variable, **depth**, in the algorithm. The aim of this is to index a path-field.

Each time we move forward in the algorithm, the depth is increased. When saving a node on the stack, the depth of this node must also be stored. Hence when popping a node from the stack, the depth is restored from the stack. See Figure 3.6.2.

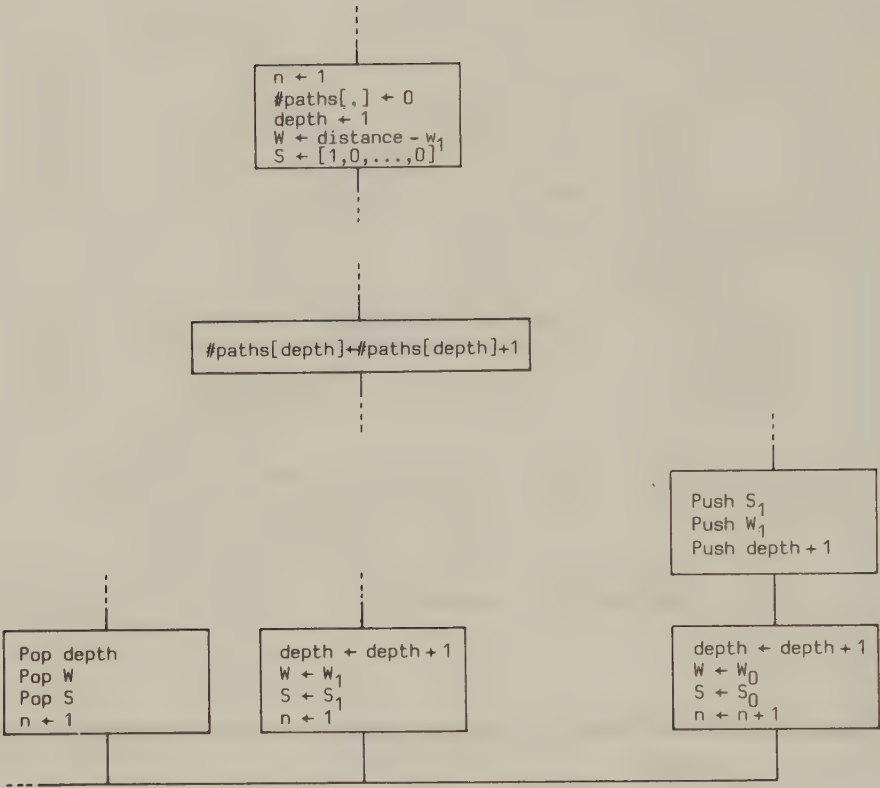


Figure 3.6.2 Modification of the FAST algorithm when length is saved.

If we use FAST together with systematic codes, then it is possible to make an additional improvement. Since $G^{(1)}(D)$ contains a large delay, equal to M time units, we have an accumulated weight in the state, W_S . Our node weight must exceed this value else we are not able to leave the tree. See Figure 3.6.3.

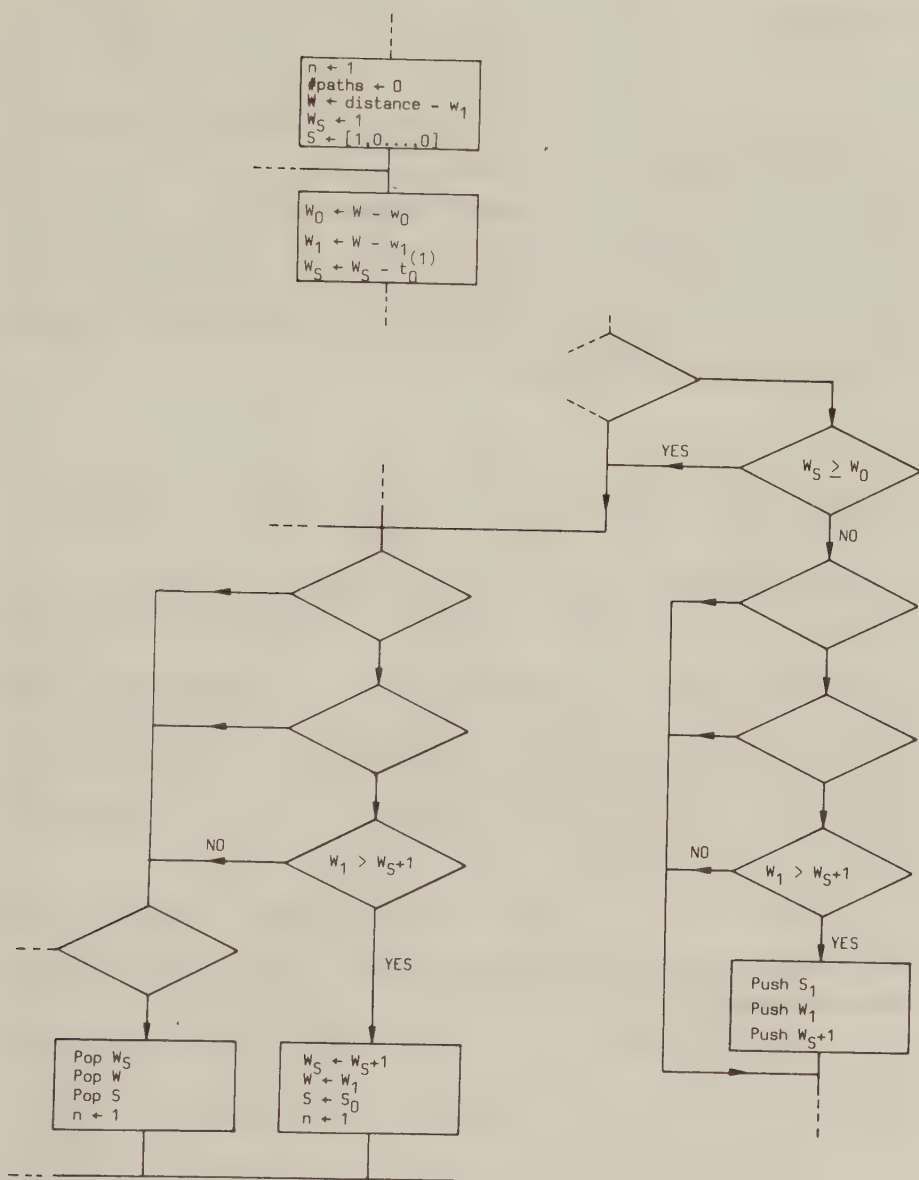


Figure 3.6.2 Modification of the FAST algorithm when used with systematic codes.

3.7 Results

The algorithm was programmed in VMS FORTRAN-77 and ran on a VAX-11/780 computer. As an example we tested a $R = 1/2$, $M = 23$ convolutional code with an optimum distance profile. It took only 20 seconds to calculate the number of paths (65) with weight $d_\infty = 26$. This code has a free distance larger than that of any previously known code of the same rate and memory. We also determined the first few values in the distance spectrum:

distance:	26	27	28	29	30	31	32
#paths:	65	0	331	0	2014	0	11359

The code generators are $G^{(1)} = 55076157$ and $G^{(2)} = 75501351$ in octal notation.

We also tested a long $R = 1/2$, $M = 68$ systematic ODP convolutional code. It took about 27 hours to calculate the first parameter in its spectrum. This code has only one path with weight $d_\infty = 31$ (28 paths with weight $d_M = 22$). The code generator is $G^{(2)} = 67114545755646670367015$.

Some years ago Massey (Massey, 1974) conjectured, in opposition to the presumed superiority of nonsystematic codes to systematic codes, that a sequential decoder will perform about as well with a systematic $R = 1/2$ code of memory $2M$ as with a nonsystematic code of memory M . Since the longer code is systematic every other channel symbol in the tail, which is used to terminate an encoded information sequence, is a beforehand known zero that can be omitted before transmission. Hence, the two codes require the same allotted space for transmission of their corresponding tails, which is the practical consequence of Massey's conjecture. To test the conjecture we have compared d_∞ for nonsystematic codes of memory M with d_∞ for systematic codes of memory $2M$:

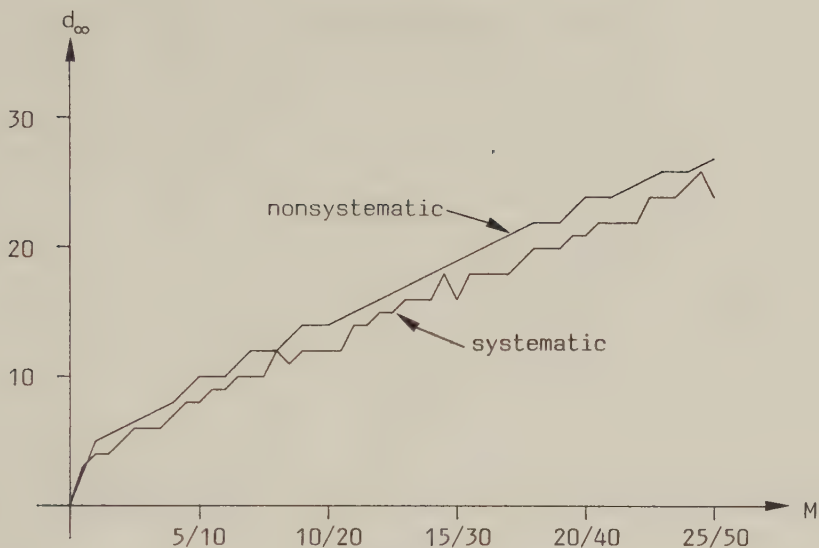


Figure 3.7.1 Systematic versus nonsystematic codes.

This figure gives a striking confirmation of Massey's conjecture. The import of Massey's conjecture, to which Figure 3.7.1 lends credence, is that a systematic $R = 1/2$ FCE can be used instead of a nonsystematic one without any sacrifice in the effective transmission rate, error probability or computational performance of a sequential decoder, provided that the memory of the systematic code is chosen as twice the allotted tail length in the information symbols. Thus, the very long systematic FCEs in the Appendix appear very attractive for use with sequential decoders. In Chapter 4 we will describe a hardware realization of FAST.

3.8 Conclusions

In this chapter we have shown how the distance profile can be utilized to limit the search to relatively small regions of error patterns. A FAST algorithm for determining spectrum parameters was described and some results were given.

An erasure.

CDC, A REALIZATION OF FAST

4.1 General problems

The total time required to find an optimum code according to some criteria increases exponentially with the code memory M . Usually when searching for optimum codes the bottle-neck is to generate the codes, cf. to generate non systematic ODP codes from a systematic code. After generating a code candidate, several tests should be made to exclude codes that will not meet necessary demands, such as non catastrophic, before FAST will be used. If not, FAST will be used in an inefficient way.

It is desirable to realize FAST as a dedicated hardware. This will have several advantages compared to a software realization. A host computer will be able to make several tests in parallel with this additional machine. It would also be possible to realize primitive operations for FAST in an efficient way and thus FAST will be even faster. All tests performed by FAST will be done in parallel.

The main obstacle in hardware realizations has been the lack of systematical methods for **all** steps from the implementation to the final machine. In this chapter we will make an attempt to bridge this gap. A way of defining a machine will be presented in Section 4.8.

We will describe these methods by realizing the **Column Distance Checker** (CDC), which is a TTL realization of FAST. Therefore some of the following sections will be attached to some existing physical components. We do not think that this will restrict the utility of our methods in a general case. In all figures that includes components, symbols in accordance with IEC publication 117-15 and 117-15A (cf. IEEE Std 91/ANSI Y32.14) are used.

4.2 Partitioning CDC

We define the complexity of a machine as the number of variables, including the state variables. Hence, it is difficult to survey a big machine. This is the reason for splitting up the implementation FAST into several logical submachines which communicate through well defined interfaces. Each submachine will be designed in a structured way right up to its realization.

Making life easier, we require that our machine is totally synchronized, e.g. one clock and one phase.

Recall Figure 3.5.3. That implementation consists of five disjoint parts, namely the path counter (#paths), the branch length counter (n), the distance profile, the actual node (state and weight) and the stack. To supervise and control these parts (slaves) we must add an additional machine (master), "The Central Scrutinizer" (TCS) (Zappa, 1979). See Figure 4.2.1. These six machines will be explained separately. Some of them may be claimed being small, but still it is a logical partition of the CDC.

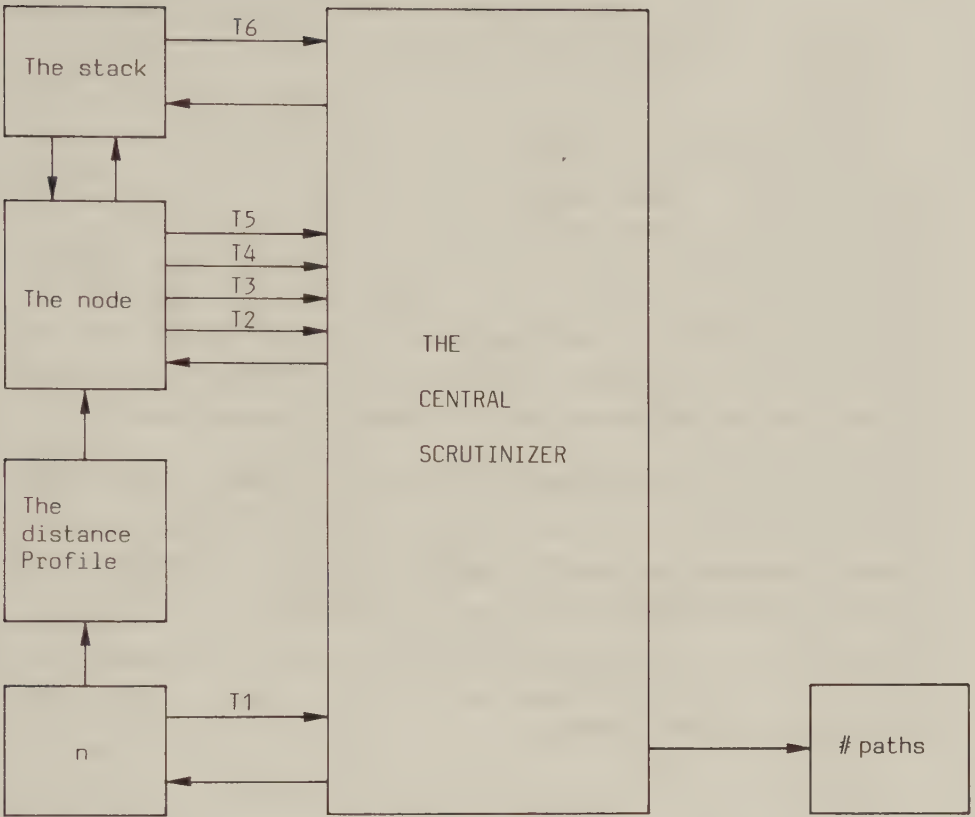


Figure 4.2.1 Submachines and control signals inside CDC.

In Figure 4.2.1 six test signals T1,T2,...,T6 are indicated. They correspond to the six different tests used by FAST. Their interpretation is given by (4.2.1)

T1	$n < M$
T2	$W_0 = 0$
T3	$W_1 \geq d_{M-1}$
T4	$W \geq d_M$
T5	$W_0 \geq d_{M-n-1}$
T6	stack empty

(4.2.1)

The logical level of each test will be specified in the following subsections. They are only valid inside this realization and thus it is better if the submachines are able to choose appropriate levels.

Our CDC must communicate with the host. These communication signals are not drawn in Figure 4.2.1. Each time we assume that a signal is external, we assign it index EXT. As an example we must have an external reset, $RESET_{EXT}$.

4.3 The path counter, #paths

The purpose of this machine is to count the number of paths with a weight equal to the tested distance. This is a simple task. There is no feedback to TCS. We must be able to perform two different operations, CLear and INCrement, according to FAST. Let us choose $2^{16}-1$, i.e. a 16 bit counter, as the maximum value of the number of paths. If we use Schottky TTL, then four 74S169s (four bit counter) will do the job. Figure 4.3.1 shows how they are interconnected.

when it is reading a distance parameter.

4.4 The branch length counter, n

FAST uses the variable n to indicate the length of the last information sequence, beginning with a one and the rest all zeros. Hence n is able to adopt a value in the set $(1, 2, \dots, M)$. When n is equal to M we must indicate it (T1). This implies that we have to be able to compare n with M . Or do we not?

If we regard n as a logical value, the variable $x = M - n$ fulfills our needs better. The comparator may be neglected. Now suppose we use a counter to represent the value of x . Assigning n the value one corresponds to loading this counter with $M - 1$. Increment n is the same as decrement x and T1 is active when x is zero. Most counters are able to detect this state.

During each loop in FAST one of two operations is performed, either load or increment (decrement x). Could these be achieved with one control signal N?

To summarize, we need a counter that is able to load a value ($= M - 1$), decrement and detect the zero state. If M is less than 256 (we hope so!), two 74S169 meet our demands.

care. The only requirement is that the CDC will solve our task. We do not need the same variables as in FAST as was the case when we selected the variable x instead of n .

4.6.1 The actual state, S

Actually we need a convolutional encoder for our purpose, but two operations must be possible on the actual state: shift and load. Load is used when we restore the state from the stack and shift when one of S_0 and S_1 is selected. See Figure 4.6.1.1.

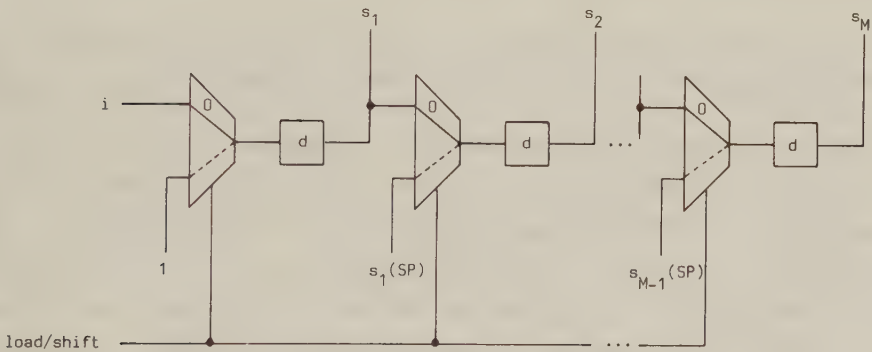


Figure 4.6.1.1 The state together with next state logic.

In Figure 4.6.1.1 we use $s_j(SP)$ as the symbol for the contents in the stack variable j . SP , stack pointer, indicates the top of the stack. The information symbol i selects either S_0 or S_1 , when we shift. The shift register 74F194 is an appropriate choice. Figure 4.6.1.2 shows two adjacent registers (the first two) realizing the state.

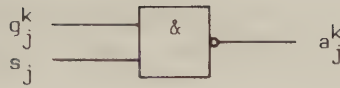


Figure 4.6.1.3 The realization of one mask bit.

The final realization of all branch symbols can be seen in Figure 4.6.1.4. We selected the nine-bit parity encoder 74F280 as the primitive component. Since $M = 40$, we have to use two levels of these. The generator symbol $g_0^{(k)}$ is constant for each calculation. Thus, if we duplicate the last level, as Figure 4.6.1.4 shows, each value for both branches will have the same delay from the inputs.

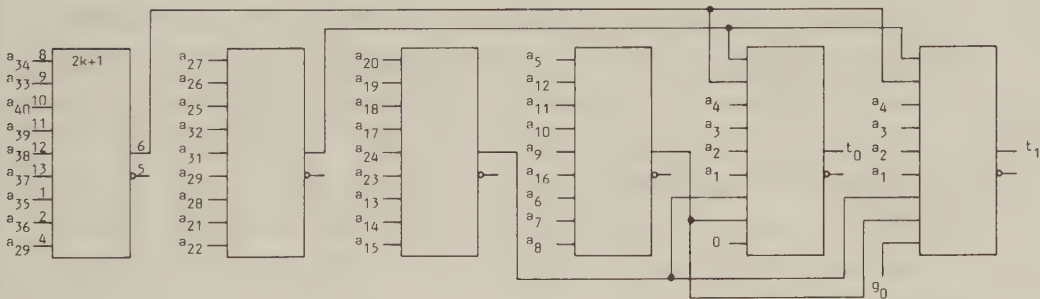


Figure 4.6.1.4 The realization of the branch symbols corresponding to one generator. Index k is left out.

4.6.2 The actual weight, W

Several tasks are connected to the weight W . First we choose the number of bits, that represents the different weights to be equal to eight. This is a practical approach, since adders are made in multiples of four bit slices and we must handle distances greater than 15 (four bits). For nonsystematic codes with $M = 25$, the free distance is 26

and we selected $M_{\max}$ to be 40. Distances greater than 255 (eight bits) are left to other authors! We choose the adder 74F283 for our purpose.

We must be able to update W depending on the outcome of the tests T3, T4 and T5. These tests will be performed here, but the decision is done by TCS. Also test T2 is a subject for this section. To summarize, we have to perform the following tests:

$$\begin{array}{ll}
 \text{T2} & W_0 = 0 \\
 \text{T3} & W_1 \geq d_{M-1} \\
 \text{T4} & W \geq d_M \\
 \text{T5} & W_0 \geq d_{M-n-1}
 \end{array}
 \tag{4.6.2.1}$$

To solve the above tasks we have to do four comparisons (equal to four comparators). Or do we not? T2 corresponds to $n = M$ and T5 to $n < M$. Thus we have a reduction in the number of components. As we said in the beginning of this Section, new variables are allowed. Let us introduce new weights called modified weights. We assign each of the old weights an index m . Each modified weight is related to the original one by a constant equal to $-d_M$. Then (4.6.2.1) becomes (4.6.2.2).

$$\begin{array}{ll}
 \text{T2} & W_{0m} = -d_M \\
 \text{T3} & W_{1m} \geq d_{M-1} - d_M \\
 \text{T4} & W_m \geq 0 \\
 \text{T5} & W_{0m} \geq d_{M-n-1} - d_M
 \end{array}
 \tag{4.6.2.2}$$

This implies that only two comparators must be used, since T4 is equal to the most significant bit (MSB) of W_m .

When $n = M$, T2 is significant. To perform this test we simply add d_M to W_{0m} . Since NAND gates with many inputs are available but not NOR gates, an all one result ($=1$) is easier to detect than a zero

result. Therefore, we add d_{M-1} instead. That settles the test T2. Subtractors are made from adders. Hence, we should save negative constants.

We are now able to specify the contents of the "distance profile". See Table 4.6.2.1.

<u>n</u>	<u>x</u>	<u>contents</u>
1	M-1	$-(d_{M-2}-d_M)$
2	M-2	$-(d_{M-3}-d_M)$
.	.	.
M-1	1	$-(d_0-d_M)$
M	0	$-(1-d_M)$

Table 4.6.2.1 The contents of the "distance profile".

Another problem is the branch weights. Each is calculated by two subtractions. See (4.6.2.3).

$$W_{im} = W_m - T_i^{(1)} - T_i^{(2)} \quad (4.6.2.3)$$

where

$$T_i^{(k)} = (0, 0, 0, 0, 0, 0, 0, t_i^{(k)})$$

and $i=0,1$ is the corresponding information symbol. But each $T_i^{(k)}$ consists of at least seven zeros, so it is not two full eight bit subtractions. The 2-complement $\tilde{T}_i^{(k)}$ is defined by

$$\begin{aligned} \tilde{T}_i^{(k)} &= (T_i^{(k)})' + 1 \\ &= (1, 1, 1, 1, 1, 1, 1, (t_i^{(k)})') + 1 \end{aligned} \quad (4.6.2.4)$$

thus (4.6.2.3) becomes

$$W_{im} = W_m + (T_i^{(1)})' + (t_i^{(2)})' \quad (4.6.2.5)$$

Connecting two add-slices together, two eight bits word and one carry input will be added. If the two words are W_m and $(T_i^{(1)})'$ and the carry input is used for $(t_i^{(2)})'$ then we have reached the end of the arithmetic. Figures 4.6.2.1 and 4.6.2.2 are the final solutions for the realizations corresponding to the tests T2, T3 and T5 and to the new node weights W_{0m} and W_{1m} .

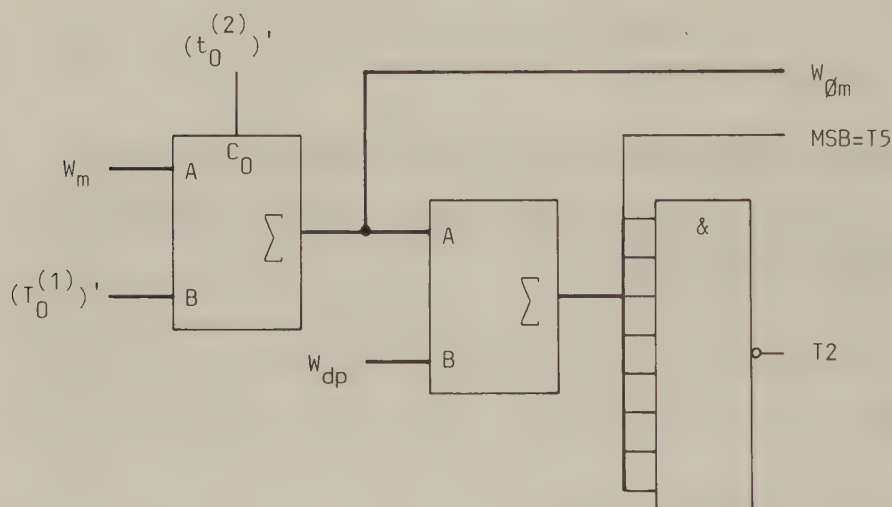


Figure 4.6.2.1 The realization of T2, T5 and W_{0m} .

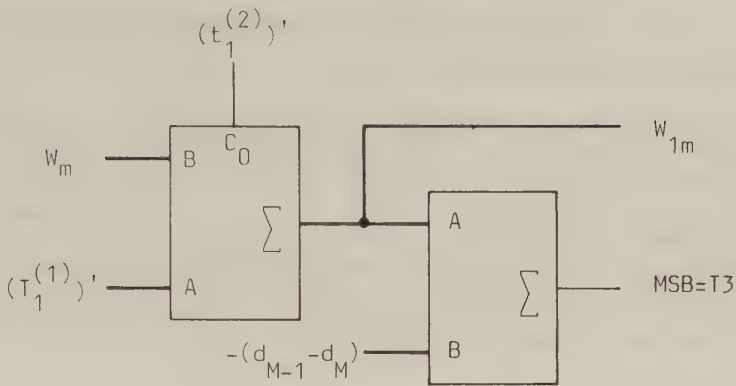


Figure 4.6.2.2 The realization of T3 and W_{1m} .

An observation should be made on these two figures. One of the problems in the CDC is to generate the parity symbols fast. We will have a relatively large delay from the result $G^{(k)}$.AND. S to the parity symbol $t^{(k)}$. Two levels of 74F280 gives 42ns (worst case). When this signal is significant, it has to pass two eight bit adders, which gives a delay of additional 60ns (30+30) before the tests are significant. In each figure the corresponding test is performed as

$$Tx \hat{=} W_{im} + \text{const}, \quad (4.6.2.6)$$

where const is either W_{dp} or $-(d_{M-1} - d_M)$. If we rewrite (4.6.2.6) as

$$Tx \hat{=} (W_m + \text{const}) + (T_i^{(1)})' + (t_i^{(2)})' \quad (4.6.2.7)$$

we see that the parity symbols just have to pass one level of addition. Thus, we save 30ns. This will be made in exchange for the cost of an additional eight bit adder in each of the Figures 4.6.2.1 and 4.6.2.2. Furthermore, in addition to the realization of (4.6.2.7) we must also realize the new node weights. These are not partial sums in (4.6.2.7). We choose the realization according to Figures 4.6.2.1 and 4.6.2.2,

since if we choose (4.6.2.7) we get two boards instead of one. It is a matter of both space and cost.

As the last part of this section we must prepare for the updating of the node weight. It must be possible to load this machine with start values, W_{EXT} . While it is running, one of the three values W_{0m} , W_{1m} and W_s (from the stack) will be selected each clockcycle. Hence one of these four values must be possible to select. TCS makes this choice. We have a flow of information (decisions) as shown in Figure 4.6.2.3.

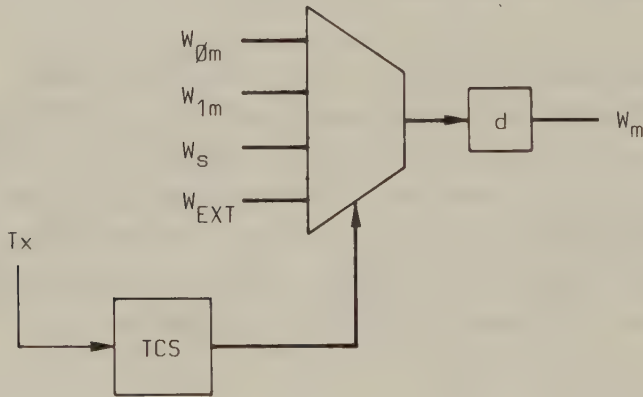


Figure 4.6.2.3 A logical scheme for updating the node weight.

As soon as TCS has made its choice, the information must pass the multiplexer before it ends up in the register. This means that we have a long loop from the node state S to the new node weight. One way to reduce it was mentioned above. Still it is possible to shorten this path, without an increase in the number of components:

Delete the multiplexer in Figure 4.6.2.3. Connect each possible weight (W_{0m} , W_{1m} and W_s) to three separate registers and the external weight (W_{EXT}) to a buffer. Each register will have an output enable signal (OE), just as the buffer has. Each enable is active low. The outputs from these four components are tied together, forming a bus.

The actual value on this bus is our node weight W_m . The choice of TCS, one of the following four-tuples (0111, 1011, 1101, 1110), will also be connected to a register. The output of this register enables one of the weights (register or buffer). Hence the delay due to the multiplexer is gone. We now have four registers (four components). The delay of the choice is eliminated. To realize the multiplexer we need four components. Hence, we saved the old register. Figure 4.6.2.4 shows our improved realization.

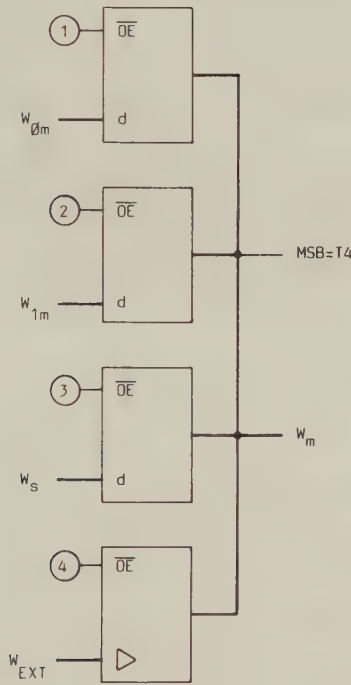


Figure 4.6.2.4 Updating the node weight.

No further task is connected to this section.

4.7 The stack

A last in first out (LIFO) queue is called a stack. We need a fast 40 bit wide stack ($M_{\max} = 40$). Using a FAST-realization on a computer, we found that to calculate d , the maximum number of nodes stored on the stack each time unit grows slightly faster than $2M$. Therefore, the depth of the stack must be slightly more than 80. Due to our choice of components we will end up with 512.

A stack can be viewed as Figure 4.7.1 shows. We have a memory and a stackpointer (SP). SP points at the last saved node (B in Figure 4.7.1). When saving a new node (C), PUSH C, the following happens. First SP is increased and then C is written into the memory location addressed by SP. This procedure may differ in different implementations of stacks. We design it to operate like this. Restoring a node, POP node, operates in the reverse order. The node assigns the contents of that memory location, addressed by SP. Then SP is decreased.

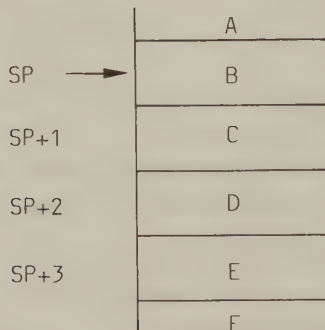


Figure 4.7.1 A stack, the stackpointer SP and a memory.

TCS chooses one of three operations (HOLD, PUSH, POP) every clockcycle. We must modify the model shown in Figure 4.7.1 to be able to do so. The two operations PUSH and POP implies that read and write must be made in parallel, in different memory locations.

Thus we split the memory as shown in Figure 4.7.2.

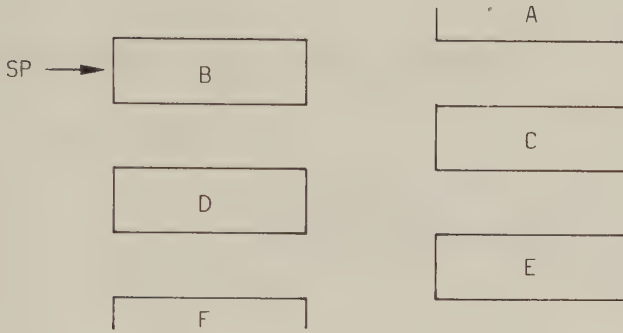


Figure 4.7.2 Parallel read and write.

The left half is stored in one memory (M_1) and the right half in another memory (M_2). Now we should have different pointers, one for each memory. Pointers are realized as counters. We label these as SC_1 and SC_2 ($SC=StackCounter$). See Figure 4.7.3.

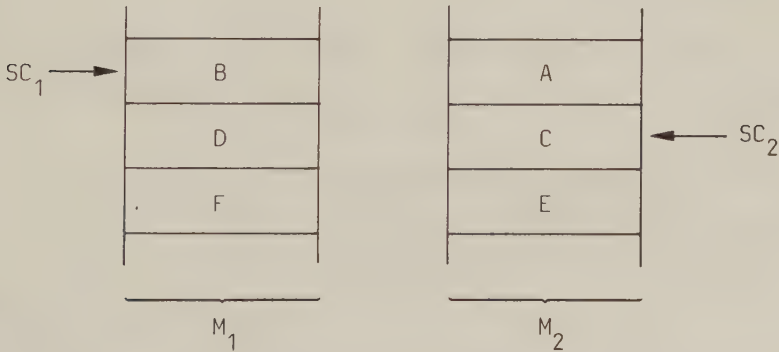


Figure 4.7.3 Symbolic scheme of the stack realization.

Assume the following. The most recently saved node on the stack is B. Let us do the following sequence of operations on the stack.

PUSH C
 PUSH D
 PUSH E

This corresponds to the following action of the counters and the memories.

write C into $M_2(SC_2)$ and increment SC_1
 write D into $M_1(SC_1)$ and increment SC_2
 write E into $M_2(SC_2)$ and increment SC_1

Each clockcycle we should operate in parallel on different components. Now it is indeed possible to do so!

Now, let us check the opposite:

POP node ; node := E
 POP node ; node := D
 POP node ; node := C

This corresponds to the following action of the counters and the memories.

read node from $M_2(SC_2)$ and decrement SC_1
 read node from $M_1(SC_1)$ and decrement SC_2
 read node from $M_2(SC_2)$ and decrement SC_1

There is no problem! We have a stack on which it is possible to perform an arbitrary operation each clockcycle. To control this we add a new machine inside the stack machine. The graph for this stack controller is shown in Figure 4.7.4.

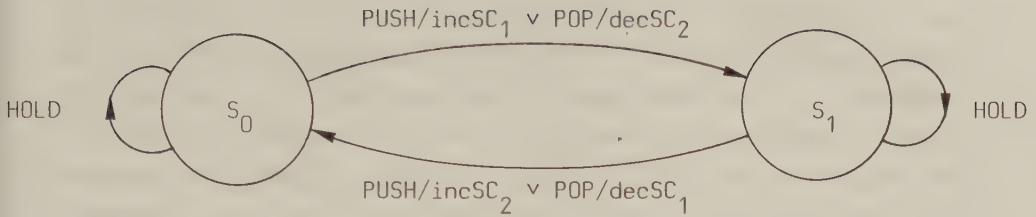


Figure 4.7.4 The stack controller.

We will not realize this machine now. It is a logical part of TCS as seen in Chapter 4.8.

The state S_0 in this machine represents that we write into memory M_2 and that we read from M_1 . The opposite is true for S_1 . Assume that we are in the state S_0 and that the counter SC_1 contains its initial value, which we select to be the all one state. This corresponds to an empty stack. Hence, if we use the realization shown in Figure 4.7.5 in conjunction with the state S_0 , the test T6 is realized.

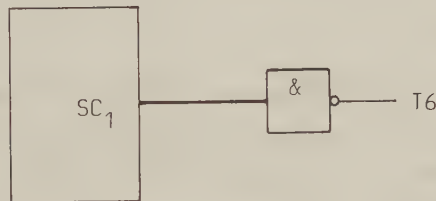


Figure 4.7.5 The realization of the test T6.

We use Am93422 memories to realize M_1 and M_2 (totally 20). Two 74S169s are used for each stack counter. If we tie the outputs from M_1 and M_2 together we get only one output from the stack. The same is true for the inputs. S_0 enables the output from M_1 and S_1 enables M_2 . To fulfil the requirement of a fully synchronized machine, we

must synchronize the write signals in the memories with our clock. For these memories we have a hold time for the address after the termination of their write signal. This time must be greater than 5ns (=typical delay time for Schottky). Since the address is generated by the counters, which are clocked, the write signal must disappear before the output from these counters changes. So if we guarantee the phase between CK and WE_1 and WE_2 , then the propagation delay time from the clock until the counter output changes, which is typically 8ns, guarantees the proper hold time. See Figure 4.7.6.

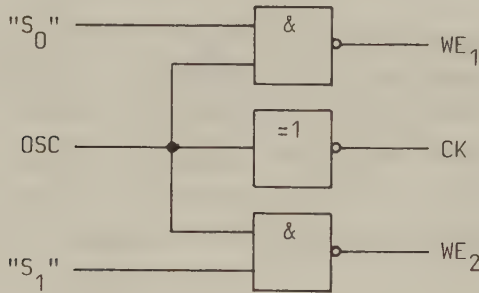


Figure 4.7.6 Write and clock synchronization. OSC is the primary clock.

Conclusion: We have realized a fast stack, able to perform an arbitrary operation from the set (HOLD, PUSH, POP). Table 4.7.7 sum up the actual action caused by each of the operations. This depends on the state, cf. Figure 4.7.4.

Actual state	Operation	Next state	Counter 1		Counter 2	
			P'	U/D'	P'	U/D'
S_0	HOLD	S_0	1	-	1	-
	PUSH	S_1	0	1	1	-
	POP	S_1	1	-	0	0
S_1	HOLD	S_1	1	-	1	-
	PUSH	S_0	1	-	0	1
	POP	S_0	0	0	1	-

Table 4.7.7 Relation between operation and actual action.

4.8 The Central Scrutinizer

So far we have realized all the surrounding machines. Their inputs and outputs are well defined. The remaining task is to connect these machines and make the whole machine, the CDC, able to operate. TCS will do this. Let us specify the inputs and outputs to and from TCS.

Input	Interpretation	Active level	Notes
RESET _{EXT}	External reset	low	
T1	$n < M$?	high	
T2	$W_0 = 0$?	low	Valid if $n = M$
T3	$W_1 \geq d_{M-1}$?	low	
T4	$W \geq d_M$?	low	
T5	$W_1 \geq d_{M-n-1}$?	low	Valid if $n < M$
T6	Stack empty ?	low	Valid in S_0

Table 4.8.1 Input signals to TCS.

A common task for the CDC is to determine d for a given code.

One way to improve FAST for this specific application was described in Section 3.6, cf. Figure 3.6.1. To realize "adjust weights in stack" in the CDC is not possible, if we have to use only one board. We decided to use only one board when a decrease in computation time was discussed, cf. (4.6.2.7). If we are interested in determining only when a tested distance is less than d , we can easily do so. Assume that we have one more machine of the type shown in Figure 4.3.2. We substitute the signal OVR with a new signal FLAG. Due to the realization of test T2 we know that if T1 is low ($n = M$) and T5 is low then we have found a path with a weight less than d . When this is the case, FLAG should be held low. Otherwise FLAG is high.

All output signals are then fixed. Table 4.8.1 is a collection of the output signals from TCS.

Output	Interpretation	Active level	Notes
INC	$\#paths = \#paths+1$	low	
FLAG	distance $< d$	low	
N	$n = 1$	low	if M-1 valid
P'_1	SC_1	low	
U/D'_1	SC_1	up=1, down=0	iff $P'_1=0$
P'_2	SC_2	low	
U/D'_2	SC_2	up=1, down=0	iff $P'_2=0$
load/shift	State action	load=1, shift=0	clear high
i	information symbol		iff shift
SEL_0	selects W_0	low	Only one
SEL_1	selects W_1	low	of $SEL_?$
SEL_s	selects W_s	low	may be
SEL_{EXT}	selects W_{EXT}	low	low!

Table 4.8.2 Output signals from TCS.

In addition to Table 4.8.2 some of the machines use a signal labeled CL or clear. These should be connected to $RESET_{EXT}$. Besides, $RESET_{EXT}$ is used to force the CDC to a state corresponding to

START in FAST. Actually we need only three states in TCS to realize FAST (START, OPERATE, STOP). OPERATE means that we loop in FAST. To include the two stack states, S_1 and S_2 , in TCS, we must split OPERATE into two new states, $OPERATE_1$ and $OPERATE_2$. Hence we have the graph shown in Figure 4.8.1.

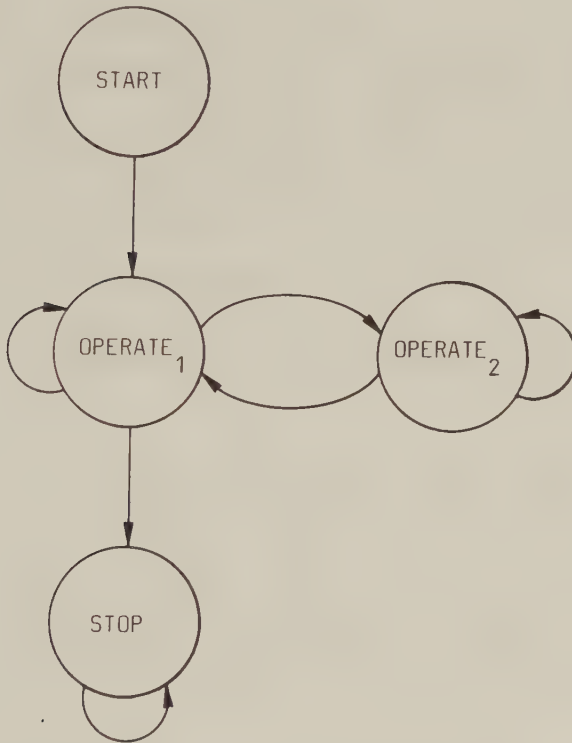


Figure 4.8.1 The states in TCS. The transitions resulting from $RESET_{EXT}$ are not drawn in the figure.

The normal procedure to realize a machine starts with defining the machine. We did this by splitting the CDC into six submachines, each except TCS is already realized. TCS tends to be the biggest machine. This is the normal case for a controller. It is important to have a clear definition of it. This is due to many aspects. It is easier to make it operate properly. If something fails, we can check where our

interpretation was wrong. The machine may be subject to a revision. This is perhaps made by an uninitiated engineer. Standardization is highly desirable. We believe that the graph approach is an efficient solution to these problems.

Once we have a graph that is functioning, this should be the input to a program that produces the realization. This is our goal. We will now describe our solution of this problem. We have a "language" that describes the activity in a graph. The realization of this was coded by Anders Hansson (Hansson, 1983). The program produces an ASCII file, which assigns output symbols to input symbols. Finally, we have written a program, MINIM, which minimizes the combinatorial part of the machine. The minimization depends highly on how we will realize the machine. Should we use PAL, PLA or common gates? Synchronous or asynchronous? MINIM takes this into consideration. The total way from our graph representation to the realization is "man-free". If the specification in the graph works, we end up with a working realization.

It is our opinion that a well specified graph is the best way to document a machine. Therefore it is desirable that we can represent a graph in a manner that is readable, both for an engineer and a computer. The procedure to describe a machine is exemplified by TCS. We must force the user of this method to produce a document, which fully describes the machine. There must not be any doubt about its function.

Specification and documentation of TCS. All the following examples deal with TCS. We use the character ";" as a statement terminator. Everything between two exclamation marks is treated as a comment. First we must specify the size of the machine. This is done as follows:


```

Number_of_states: 4;
Number_of_inputs: 7;      ! Tests and RESETEXT. !
Number_of_outputs: 13;

```

If we only want to specify a set of switching functions (a combinatorial circuit) then the number of states is equal to one!

Now when the cardinality of each set is specified, the sets themselves should be declared in a symbolic form. A symbolic name is a string of letters, digits and underscores "_". We use this overhead only for checking. Enumeration of elements in each set is delimited with commas ",". When we specify an input or an output signal, a position must also be specified. This is done by assigning a one or a zero to that position. The rest will have "-".

States: START, OPERATE_1, OPERATE_2, STOP;

Inputs: RESET = 0-----,

T1 = -1-----,

T2 = --0-----,

T3 = ---0----,

T4 = ----0--,

T5 = -----0-,

T6 = -----0;

Outputs: INC = 0-----,

FLAG = -0-----,

N = --0-----,

P_1 = ---0-----,

UP_1 = ----1-----,

P_2 = -----0-----,

UP_2 = -----1-----,

LOAD = -----1-----,

I = -----1-----,

SEL_0 = -----0-----,

SEL_1 = -----0-----,

SEL_S = -----0-----,

SEL_E = -----0-----;

Logical combinations of input or output elements are natural to specify. In the stack case we operate with HOLD, PUSH and POP. It is not interesting to know the actual control signal. For this purpose let us introduce mnemonics for combinations of signals in the same set. We use "*" for concatenation and "~" for negation. As soon as a mnemonic is defined then we may use it in new mnemonics, with one exception. We are not allowed to negate a mnemonic. In later implementations this may be possible. The limitation was made to reduce the complexity of the interpreter.

Mnemonics:

```

HOLD    = P_1' * P_2',          ! HOLD    = ---1-1----- !
PUSH_1  = P_1 * UP_1 * P_2',    ! PUSH_1  = ---011----- !
POP_1   = P_1' * P_2 * UP_2',   ! POP_1   = ---1-00----- !
PUSH_2  = P_1' * P_2 * UP_2,    ! PUSH_2  = ---1-01----- !
POP_2   = P_1 * UP_1' * P_2',   ! POP_2   = ---001----- !
SHIFT   = LOAD',                ! SHIFT   = -----0----- !
W_0     = SHIFT*I'*SEL_0*SEL_1'*SEL_S'*SEL_E',
W_1     = HOLD*SHIFT*I'*SEL_0'*SEL_1*SEL_S'*SEL_E',
W_S_1   = POP_1*LOAD*SEL_0'*SEL_1'*SEL_S*SEL_E',
W_S_2   = POP_2*LOAD*SEL_0'*SEL_1'*SEL_S*SEL_E',
W_E     = HOLD*SHIFT*I'*SEL_0'*SEL_1'*SEL_S'*SEL_E;

```

Each state must be assigned a binary value, the coding of the machine. There are two ways of coding: A direct specification (we specify the code) or autocoding (the interpreter of the document assigns the code). Autocoding is done in one of the following ways. If it is a synchronous machine we first investigate if an SP-partition (full or partial) (Lee, 1978) is possible or not. If not, each state gets the natural binary code for the order in which it was declared. For the synchronous case the type of flip-flops must be specified. For an asynchronous machine the program tries to specify a code. If it can not, an error message must occur.

```

State_assignment: START      = 00,
                    OPERATE_1= 01,
                    OPERATE_2= 11,
                    STOP      = 10;
                    ! Autocode is possible. !

```

```

Machine_type:    Synchronous;
                    ! else Asynchronous !

```

Flipflops: $Q1 = JP, Q0 = JP;$
 ! Alternatively: $All = JP (=JK')$. !
 ! Also available. JK, SR, D and T. !

Each switching function will be realized with a combinatorial circuit. The minimization of a set of functions depends both on the type of components and on the synchronous/asynchronous choice. As the last part of the declarations, the type of components must be specified.

Components: MMI;
 ! =PAL, but each function is negated. !
 ! Choices: DISCRETE, PLA, PAL and MEMORY. !
 ! If MEMORY, then no minimization will be done. !
 ! If MEMORY and ASYNCHRONOUS, then ERROR. !

This was the end of the declarations. The statement **State=** acts both as an end of declarations and as a specification of the present state. We now have the task to specify the implementation of this machine. The interpreter must check that each statement is legal. As an example, if we specify the input as $HOLD * POP_1$, then we must have an error indication. P_2 can not be one (HOLD) and zero (POP_1) at the same time.

To make a statement that corresponds to a transition in the graph we use the following notation:

input/output,next-state;

If two input specifications (which cover a common input tuple) try to specify different outputs, zero or one, then the interpreter should react in one of the following ways: If we have specified **Nowarnings;** then the last value should be used, else an error message will occur.

When we specify the performance of the machine three additional statements will facilitate the specification:

The first statement is:

Unspecified combinations: O,S;

For every output combination that is not subject to an assignment this (O,S) is valid. Hence, we are only permitted to do this once per state. If either the actual output O or the next state S is a blank (not stated) then we treat this as a don't care (-). Whenever the statement "State=" is given, the unspecified combinations are treated as don't cares until something else is specified.

The second statement concerns the input. When we have many input symbols, many input specifications tend to have a common factor (I), i.e. most of the input symbols are equal for a sequence of statements. To exclude this factor from each specification we simply write:

Default_input: I;

After this statement, each specified input combination will take this as default (until the next Default_input is stated). When evaluating the actual input signal we use the last rule that was stated.

```
Example: Default_input: 0-1-10;  
-10-11/output,next-state;
```

This is equivalent to: 010-11/output,next-state;

The last statement acts as the second one, but on the output.

Default_output: 0,5;

We are now able to specify the behavior of the machine. Recall Figure 3.5.3. This is the implementation which we now realize by doing the following specification.

Nowarnings;

!-----!

State= OPERATE_1;**Unspecified_combinations:** ,START; ! Due to RESET !**Default_input:** RESET'*T1*T5;**Default_output:** INC'*FLAG'*N'*W_O;

/HOLD,OPERATE_1;

T3*T4/PUSH_1,OPERATE_2; ! Error if not Nowarnings. !

Default_input: RESET'*T1*T5';**Default_output:** INC'*FLAG';

T6/ ,STOP;

T6'/N*W_S_1,OPERATE_2;

T3*T4/N*W_1,OPERATE_1;

Default_input: RESET'*T1'*T2;**Default_output:** INC*FLAG';

T6/ ,STOP;

T6'/N*W_S_1,OPERATE_2;

T3*T4/N*W_1,OPERATE_1;

Default_input: RESET'*T1'*T2'*T5';**Default_output:** INC'*FLAG';

T6/ ,STOP;

T6'/N*W_S_1,OPERATE_2;

T3*T4/N*W_1,OPERATE_1;

Default_input: RESET'*T1'*T2'*T5;**Default_output:** INC'*FLAG;

T6/ ,STOP;

T6'/N*W_S_1,OPERATE_2;

T3*T4/N*W_1,OPERATE_1;

!-----!

State= OPERATE_2;

RESET/ ,START;

Default_input: RESET'*T1*T5;

Default_output: INC'*FLAG'*N'*W_O;

/HOLD,OPERATE_2;

T3*T4/PUSH_1,OPERATE_1;

Default_input: RESET'*T1*T5';

Default_output: INC'*FLAG';

T6'/N*_S_1,OPERATE_1;

T3*T4/N*_1,OPERATE_2;

Default_input: RESET'*T1'*T2;

Default_output: INC*FLAG';

T6'/N*_S_1,OPERATE_1;

T3*T4/N*_1,OPERATE_2;

Default_input: RESET'*T1'*T2'*T5';

Default_output: INC'*FLAG';

T6'/N*_S_1,OPERATE_1;

T3*T4/N*_1,OPERATE_2;

Default_input: RESET'*T1'*T2'*T5;

Default_output: INC'*FLAG;

T6'/N*_S_1,OPERATE_1;

T3*T4/N*_1,OPERATE_2;

!-----!

State= START;

Unspecified_combinations: ,START;

RESET'/INC'*FLAG'*N*_E,OPERATE_1;

!-----!

State= STOP;

Unspecified_combinations: ,START;

RESET'/INC'*FLAG',STOP;

The idea is that the above specification should be the last manual step in a realization. After this step, the programs produce the desired form of realization. Table 4.8.3 shows an intermediate result in this procedure. From this result, programmed PALs are produced automatically!

The following table contains a list of selected prime implicants for this synchronous-PAL case.

X indicates a chosen prime implicant.

Variables ----->								Functions ----->																	
8	7	6	5	4	3	2	1	0	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
0	-	-	1	-	0	0	0	-	.	.	.	X	.	.	.	.	.	.	.	.	.	.	.	.	.
1	-	-	1	-	0	0	0	-	.	.	.	.	.	X	.	.	.	.	.	.	.	.	.	.	.
-	-	1	0	-	0	0	-	-	.	.	.	.	.	.	.	.	.	.	X	.	.	.	X	.	.
-	1	-	0	-	0	0	-	-	.	.	.	.	.	.	.	.	.	.	X	.	.	.	.	.	.
-	-	1	-	-	0	0	1	-	.	.	.	.	.	.	.	.	.	.	X	.	.	.	X	.	.
-	1	-	-	-	0	0	1	-	.	.	.	.	.	.	.	.	.	.	X	.	.	.	.	.	.
-	1	-	0	1	-	-	0	-	.	X	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
-	-	1	1	-	-	1	0	-	.	.	.	.	.	.	.	.	.	.	.	.	.	.	X	.	.
-	-	1	1	-	1	-	0	-	.	.	.	.	.	.	.	.	.	.	.	.	.	.	X	.	.
-	1	-	0	-	-	1	-	1	.	.	.	.	.	.	.	.	.	.	.	X	.	.	.	.	.
-	1	-	0	-	1	-	-	1	.	.	.	.	.	.	.	.	.	.	.	X	.	.	.	.	.
-	1	-	-	-	-	1	1	1	.	.	.	.	.	.	.	.	.	.	.	X	.	.	.	.	.
-	1	-	-	-	1	-	1	1	.	.	.	.	.	.	.	.	.	.	.	X	.	.	.	.	.
-	-	-	0	-	0	0	-	-	.	.	.	.	.	.	.	.	.	.	.	.	.	X	.	.	.
-	-	-	-	-	0	0	1	-	.	.	.	.	.	.	.	.	.	.	.	.	.	X	.	.	.
0	-	-	0	-	-	1	-	-	.	.	.	.	X	.	.	.	.	.	.	.	.	.	.	.	.
0	-	-	0	-	1	-	-	-	.	.	.	.	X	.	.	.	.	.	.	.	.	.	.	.	.
-	-	1	-	-	0	0	-	-	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	X
-	1	-	0	0	-	-	-	-	X	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
0	-	-	-	-	-	1	1	-	.	.	.	.	X	.	.	.	.	.	.	.	.	.	.	.	.
0	-	-	-	-	1	-	1	-	.	.	.	.	X	.	.	.	.	.	.	.	.	.	.	.	.
-	-	-	1	-	-	1	0	-	.	.	.	.	.	.	.	.	.	.	.	.	.	X	.	.	.
-	-	-	1	-	1	-	0	-	.	.	.	.	.	.	.	.	.	.	.	.	.	X	.	.	.
-	-	1	1	-	-	-	0	-	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	X
-	1	-	-	-	-	1	-	0	.	.	.	.	.	.	.	.	.	X	.	.	.	.	.	.	.
-	1	-	-	-	1	-	-	0	.	.	.	.	.	.	.	.	.	X	.	.	.	.	.	.	.
-	1	-	1	-	-	-	0	-	.	.	.	.	.	.	.	.	X	X	.	.	.	.	.	.	.
1	-	-	0	-	-	1	-	-	.	.	.	X	.	.	.	.	.	.	.	.	.	.	.	.	.
1	-	-	0	-	1	-	-	-	.	.	.	X	.	.	.	.	.	.	.	.	.	.	.	.	.
1	-	-	-	-	-	1	1	-	.	.	.	X	.	.	.	.	.	.	.	.	.	.	.	.	.
1	-	-	-	-	1	-	1	-	.	.	.	X	.	.	.	.	.	.	.	.	.	.	.	.	.
-	-	-	-	-	0	0	-	-	.	.	.	.	.	.	.	X	.	.	.	.	.	.	.	.	.
-	-	-	1	-	-	-	0	-	.	.	.	.	.	.	.	X	.	.	.	.	.	.	.	.	.
-	0	1	-	-	-	-	-	-	.	.	.	.	.	.	.	.	.	.	.	.	.	X	.	.	.
-	-	1	-	-	-	-	-	1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	X
-	0	-	-	-	-	-	-	-	.	.	X	.	.	.	.	X	.	.	.	X	X	.	.	.	.
-	-	-	0	-	-	-	-	-	.	.	X	.	.	.	.	.	.	.	.	.	.	.	.	.	.
0	-	-	-	-	-	-	-	-	.	.	.	.	.	X	.	.	.	.	.	.	.	.	.	.	.
-	-	0	-	-	-	-	-	-	.	.	.	.	.	.	.	.	.	.	.	.	.	X	.	X	.
-	-	-	-	-	-	-	1	-	.	.	X	.	.	.	.	.	.	.	.	.	.	.	.	.	.
1	-	-	-	-	-	-	-	-	.	.	.	.	X	.	.	.	.	.	.	.	.	.	.	X	.

Table 4.8.3 Result from MINIM.

4.9 Conclusions

We believe that it is important to decompose the problem in a way that suits the structure of the problem, without any preconceptions about the way it will be realized. The latter is unfortunately the most common case and consequently many constructions suffer from this. The concept **Random logic** refers to this latter group and many well done realizations carry this deceptive name.

APPENDIX

Table A.1 Minimum distances for systematic ODP codes in the range $1 \leq M \leq 96$. For each memory the code with the fewest paths is given.

M	d_M	$\#_M$	$G^{(2)}$
1	3	1	6
2	3	1	6
3	4	3	64
4	4	1	64
5	5	5	65
6	5	2	650
7	6	11	670
8	6	5	670
9	6	1	6710
10	7	12	6710
11	7	5	6711
12	8	29	67114
13	8	12	67114
14	8	6	67115
15	8	1	671150
16	9	18	671144
17	9	7	671151
18	9	3	6711514
19	10	31	6711454
20	10	13	6711454
21	10	4	67114544
22	10	1	67115142
23	11	27	67114543
24	11	11	671145430
25	11	5	671151572
26	11	1	671151505

Table A.1 cont.

M	d_M	$\#_M$	$G^{(2)}$
27	12	21	6711454574
28	12	8	6711454306
29	12	2	6711454311
30	13	43	67114545754
31	13	15	67114545754
32	13	4	67114545755
33	13	1	671145457554
34	14	34	671145457556
35	14	14	671145454470
36	14	5	6711454544704
37	14	2	6711454544676
38	15	31	6711454575564
39	15	12	67114545755644
40	15	3	67114545755712
41	15	1	67114545755713
42	16	31	671145457556464
43	16	14	671145457556464
44	16	5	671145457556153
45	16	1	6711454575561314
46	17	39	6711454575564666
47	17	13	6711454575564667
48	17	4	67114545755646674
49	17	1	67114545755646676
50	18	38	67114545755646676
51	18	16	671145457556466760
52	18	7	671145457556466760
53	18	2	671145457550027077
54	19	43	6711454575571301174
55	19	20	6711454575571301176
56	19	7	6711454575571301176
57	19	2	67114545755713011760
58	20	60	67114545755713011760
59	20	25	67114545755646670367
60	20	10	671145457556466703670
61	20	2	671145457557130117610
62	20	1	671145457557130117611
63	21	25	6711454575571301176114
64	21	10	6711454575571301176114
65	21	2	6711454575571301176114
66	22	71	67114545755713011761144
67	22	29	67114545755646670367016
68	22	9	67114545755646670367017
69	22	4	671145457556466703670170
70	22	1	671145457556466703670170
71	23	46	671145457557130117611463
72	23	16	6711454575564667036701444
73	23	5	6711454575564667036701446

Table A.1 cont.

M	d_M	$\#_M$	$G^{(2)}$
74	23	2	6711454575564667036701447
75	24	56	67114545755713011761146370
76	24	20	67114545755713011761146342
77	24	8	67114545755713011761146373
78	24	3	671145457557130117611463424
79	25	74	671145457557130117611463432
80	25	33	671145457557130117611463433
81	25	16	6711454575571301176114634334
82	25	4	6711454575564667036701447272
83	25	1	6711454575564667036701447277
84	26	41	67114545755646670367014472730
85	26	20	67114545755713011761146343362
86	26	6	67114545755713011761146343363
87	26	2	671145457557130117611463433634
88	27	62	671145457556466703670144727304
89	27	28	671145457556466703670144727305
90	27	11	6711454575564667036701447273054
91	27	5	6711454575564667036701447273056
92	27	1	6711454575564667036701447273357
93	28	42	67114545755646670367014472730510
94	28	20	67114545755646670367014472730512
95	28	5	67114545755646670367014472730511
96	28	1	671145457556466703670144727305110

Table A.2 Free distances for systematic ODP codes in the range $1 \leq M \leq 60$. For each memory the code with the fewest paths is given.

M	d_M	$\#_M$	d_∞	$\#_\infty$	$G^{(2)}$
1	3	2	3	1	6
2	3	1	4	2	7
3	4	3	4	1	64
4	4	1	5	2	72
5	5	5	6	3	73
6	5	3	6	2	654
7	5	3	6	2	654
8	6	5	7	2	715
9	6	5	8	5	6714
10	7	13	8	3	7152
11	7	8	9	5	7155
12	8	29	9	1	67114
13	8	17	10	5	67116
14	8	6	10	4	67115
15	8	3	10	3	671144
16	9	22	12	13	671166
17	9	13	11	4	714447
18	9	7	12	5	7144474
19	10	31	12	3	7144616
20	10	18	12	2	6711455
21	10	7	12	2	67115144
22	10	8	14	12	67114552
23	11	27	14	10	71446165
24	11	16	15	7	671145434
25	11	7	15	6	671145452
26	11	2	16	16	714476125
27	12	24	16	10	6711455364
28	12	18	16	11	7144761242
29	12	6	18	22	7144760535
30	13	55	16	3	67114545644
31	13	24	18	11	67114543066
32	13	10	18	12	71447605245
33	13	3	18	9	714461625304
34	14	42	18	1	714461625306
35	14	19	19	2	714461626555
36	14	15	20	21	7144616573444
37	14	6	20	13	7144616253122
38	15	37	20	5	7144616573453
39	15	19	21	10	67114545755614
40	15	16	21	6	71446162655566
41	15	5	22	16	71446162531365
42	16	44	22	6	714461626554424
43	16	19	22	13	671145454470302
44	16	11	22	9	714461625307621
45	16	4	24	18	6711454575561514

Table A.2 cont.

M	d_M	$\#_M$	d_∞	$\#_\infty$	$G^{(2)}$
46	17	41	24	16	7144616265556136
47	17	27	24	5	6711454306444073
48	17	11	25	5	71446162655442554
49	17	5	26	28	67114545755613152
50	18	53	24	3	67114545755002707
51	18	23	26	16	671145457556131564
52	18	10	26	10	714461626554010456
53	18	4	27	8	714461626554010475
54	19	49	26	2	7144616265540104574
55	19	30	27	3	7144616265540104742
56	19	15	27	2	7144616265540104741
57	19	7	28	7	67114545755646670354
58	20	70	27	2	67114545755646670366
59	20	32	28	5	71446162655401045745
60	20	15	30	12	671145457557130117614

Table A.3 The first spectrum parameters for nonsystematic ODP codes in the range $2 \leq M \leq 25$. For each memory the code with the fewest paths is given except for those marked with a star. The notation $\#j$ means the number of paths having weight $d_{\infty}+j$.

M	$G^{(1)}$	$G^{(2)}$	d_{∞}	#0	#1	#2	#3	#4
2	7	5	5	1	2	4	8	16
3	74	54	6	1	3	5	11	25
4	62	56	7	2	3	4	16	37
5	75	55	8	2	7	10	18	49
6	634	564	10	12	0	53	0	234
7	626	572	10	1	6	13	20	64
8	751	557	12	10	9	30	51	156
9	7664	5714	12	1	8	8	31	73
10	7512	5562	14	19	0	80	0	450
11	6643	5175	14	1	10	25	46	105
12	63374	47244	15	2	10	29	55	138
13	45332	77136	16	5	15	21	56	161
14	65231	43677	17	3	16	44	62	172
15	517604	664134	18	10	0	86	0	417
16	717066	522702	19	9	16	48	112	259
17	506477	673711	20	12	37	47	140	358
18	5653664	7746714	21	13	38	63	142	363
19	5122642	7315626	22	26	0	160	0	916
20	6567413	5322305	22	2	28	51	86	222
21	67520654	50371444	24	40	0	251	0	1379
*22	67132702	50516146	24	25	0	163	0	844
23	55076157	75501351	26	65	0	331	0	2014
*24	673275374	506302644	26	26	0	182	0	940
*25	665041116	516260772	27	24	54	125	278	637

REFERENCES

- Bahl et al, 1968** L. R. Bahl, C. D. Cullum, W. D. Frazer and F. Jelinek, "An efficient algorithm for computing the free distance," IEEE T-IT, vol IT-18, pp. 437-439, May 1972.
- Blahut, 1983** Richard E. Blahut, "Time and Frequency Domain Processing", The Impact of Processing Techniques on Communications, NATO Advanced Study Institute, Chateau de Bonas (Gers) France, 11-22 JULY 1983
- Bussgang, 1965** Julian J. Bussgang, "Some Properties of Binary Convolutional Code Generators", IEEE T-IT, pp. 90-100, 1965.
- Chevillat et al, 1976** P. R. Chevillat and Daniel J. Costello, Jr., "Distance and computation in sequential decoding", IEEE Trans. Commun., vol COM-24, pp. 440-447, April 1976.
- Costello, 1969** Daniel J. Costello, Jr., "Construction of Convolutional Codes for Sequential Decoding", Technical Report EE-692, Dept. of Electrical Engineering, University of Notre Dame, 1969.

- Divsalar, 1978** Dariush Divsalar, "Performance of Mismatched Receivers on Bandlimited Channels", Ph.D. dissertation, Univ. California, Los Angeles, 1978.
- Fano, 1963** R.M. Fano, "A heuristic discussion of probabilistic decoding", IEEE Trans. Inform. Theory, vol. IT-9, pp. 64-74, Apr. 1963.
- Gallager, 1968** R.G. Gallager, Information Theory and Reliable Communication. New York: Wiley, 1968.
- Hamming, 1950** R. W. Hamming, "Error Detecting and Correcting Codes", Bell System Tech. J., 29, pp. 147-160, 1950
- Hansson, 1983** Anders Hansson, "Första delen av en kiselkompilator för automatisk generering av sekvensnät", in swedish, Lund, 1983
- Johannesson, 1975** R. Johannesson, "Robustly-optimal rate one-half binary convolutional codes", IEEE Trans. Inform. Theory, vol. IT-21, pp. 464-468, July 1975.
- Larsen, 1973** Comments on "An efficient algorithm for computing free distances" by Bahl et al.; T-IT Jul 1973 page 577-579.
- Lee, 1978** Samuel C. Lee, "Modern Switching Theory and Digital Design", Printice-Hall, Inc., Englewood Cliffs, New Jersey, 07632, 1978.
- Massey, 1963** James L. Massey, "Threshold Decoding", Cambridge, Mass.: The M.I.T. Press, 1963.
- Massey et al, 1967** James L. Massey and M. K. Sain, "Codes,

- Automata, and Continuous Systems: Explicit Interconnections", IEEE Trans. Automatic Control, AC-12, pp. 644-650, 1967.
- Massey et al, 1968** James L. Massey and M. K. Sain, "Inverses of Linear Sequential Circuits", IEEE Trans. on Computers, C-17, pp. 330-337, 1968.
- Massey et al, 1971** James L. Massey and D. J. Costello, Jr., "Nonsystematic convolutional codes for sequential decoding in space applications", IEEE Trans. Commun. Tech., vol COM-19, Part II, pp. 806-813, Oct. 1971.
- Massey, 1974** James L. Massey, private communication with Rolf Johannesson, 1974.
- Meeberg, 1974** L. Van de Meeberg, "A tightened upper bound on the error probability of binary convolutional codes with Viterbi decoding", IEEE Trans. Inform. Theory, vol. IT-20, pp. 389-391, May 1974.
- Odenwalder, 1970** J. P. Odenwalder, "Optimal Decoding of Convolutional Codes", Ph.D. dissertation, Dept. Syst. Sci., Sch. Eng. Appl. Sci., Univ. California, Los Angeles, 1970.
- Post, 1977** K.A. Post, "Explicit evaluation of Viterbi's union bounds on convolutional code performance for the binary symmetric channel", IEEE Trans. Inform. Theory, vol. IT-23, pp. 403-404, May 1977.
- Post, 1980** K.A. Post, "New bounds on the performance of binary convolutional codes using Viterbi decoding on a binary symmetric channel",

Proceedings of Symposium over Informatietheorie in de Benelux, Zoetermeer (Nederland), May 29-30, 1980.

- Reed et al, 1960** I. S. Reed and G. Solomon, "Polynomials Codes over Certain Finite Fields," J. Soc. Indust. Appl Math., *, pp. 300-304, 1960
- Schalkwijk et al, 1979** J.P.M. Schalkwijk, K.A. Post, and J.P.J.C. Aarts, "On a method of calculating the event error probability of convolutional codes with maximum likelihood decoding", IEEE Trans. Inform. Theory, vol. IT-25, pp. 737-743, Nov. 1979.
- Shannon, 1948** Claude E. Shannon, "A Mathematical Theory of Communication", Bell System Tech. J., vol 27 (pt. I), pp. 379-423, 1948."
- Viterbi, 1971** A.J. Viterbi, "Convolutional codes and their performance in communication systems", IEEE Trans. Commun. Technol., vol. COM-19, pp. 751-772, Oct. 1971.
- Viterbi et al, 1979** A.J. Viterbi and J.K. Omura, Principles of Digital Communication and Coding. New York: McGraw-Hill, 1979.
- Zappa, 1979** Frank Zappa, "Joe's Garage, Act I.", CBS 86101, 1979.

Mats Cedervall

CONTRIBUTIONS TO THE DECODING AND STRUCTURE OF CONVOLUTIONAL CODES.

Error control coding is often used to increase the efficiency of data communication systems. This is due to the dramatic improvement in error control techniques and the increased use of satellite links.

There are basically two different approaches to the problems of constructing good codes. The algebraists choose block codes with suitable mathematical structure to simplify the development of decoding algorithms. Rather than dealing with specific codes the probabilists specify decoding algorithms which are efficient procedures for certain ensembles of randomly selected tree codes. Convolutional codes constitute the class of linear, time-invariant tree codes. In almost every error correcting application convolutional codes are superior to block codes.

This thesis presents methods for determining quality parameters for convolutional codes, viz. the decoding error probability and different distances. We give bounds on decoding error probability which are tighter than previously known bounds. These bounds are useful when we compare different codes with each other. FAST algorithms for computing distance spectrum parameters are presented. They are easily realized as computer programs. Finally, a structured way of specifying a sequential machine is described.

Previously published theses at the Department of Computer Engineering, University of Lund:

Bertil Svensson: LUCAS Processor Array - Design and Applications, PhD thesis 1983.

Christer Fernström: The LUCAS Associative Array Processor and its Programming Environment, PhD thesis 1983.

Ivan Kruzela: An Associative Array Processor Supporting a Relational Algebra, PhD thesis 1983.